A Neural Network Based Approach to the Control of Flexible-Joint Manipulators

Vladimir Zeman

A Thesis

in

the Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Engineering at

Concordia University

Montréal, Québec, Canada

March 1991

ISBN 0-315-64726-4

ABSTRACT

A Neural Network Based Approach to the Control of Flexible-Joint Manipulators

V. Zeman

Traditional robot control strategies assume both joint and link rigidity for the purpose of simplifying the control problem. The demand for greater precision coupled with the increased use of lightweight materials necessitates the inclusion of elastic dynamics in the control strategy. These highly nonlinear dynamics which increase the order of the system are extremely difficult to formulate with sufficient accuracy. The standard form of adaptive control does not appear to be applicable since the basic assumptions on the system dynamics and nonlinear characteristics are rarely satisfied. We propose an alternate control scheme which does not rely on accurate *a priori* knowledge of the manipulator dynamics, but instead can "learn" these dynamics by using a neural network.

A backpropagation network is trained off-line to model the inverse-dynamics mapping of the manipulator. The trained network is then inserted in the manipulator's control system wherein its purpose is to linearize the robot's dynamics. The overall system response can then be selected independently of the robot's dynamics, by specifying a set of servo feedback gain constants. Simulations for a single-link flexible-joint manipulator illustrate the performance of the resulting closed-loop control system, and reveal some of the practical issues involved in neural network training and control.

Our control strategy is similar in structure to standard feedback linearization, with two important differences. First, the feedback signal is not constrained to entering the robot inputs linearly, as it is in feedback linearization, therefore our system does not preclude the linearizability of certain flexible-joint configurations which have been shown to be non-linearizable by the conventional approach. Second, the use of a neural network obviates the need for any prior knowledge of the manipulator's dynamical equations, requiring only an accurate estimate of the order of the system.

In memory of my mother, Eva.

ACKNOWLEDGEMENTS

I would like to express my sincerest thanks to my thesis supervisor, Professor Patel, who not only provided much needed guidance and encouragement, but most importantly, it was his continual insistence on "digging deeper" that finally resulted in a work that I can take pride in. Many thanks, also, to Professor Khorasani for taking the time to bounce ideas back and forth, and for his relentless attention to detail.

My time at Concordia has coincided with some personal up's and down's, and external factors have made it very tempting to leave the program on more than one occasion. I am very fortunate to have had the complete confidence of knowing that my father was behind me the entire way, and that at those times that I needed something to fall back on, he provided the safety net.

Only my wife, Damienne, could know when and how to provide the inspiration to make me persevere, and when to substitute a swift kick to the *derrière*. She did both with great love and grace.

It has been a pleasure to work with the Electrical and Computer Engineering staff and graduate students. Special thanks go to Henry, Nicky, Dave, Guy and Maureen, who were willing to go the extra mile.

Vlad Zeman, April 1991.

CONTENTS

List of Figures

List of Tables

1 INTRODUCTION

The aim of this research work is to design a system to control a flexible joint manipulator with unknown dynamics. Every real manipulator has some degree of both joint and link elasticity, but these effects are neglected for most industrial robots. In certain cases, however, the assumption of rigidity results in insufficient control accuracy and can even cause system instability. For example, lightweight robots with long thin links, such as the Canadarm aboard NASA's Space Shuttle have a relatively high degree of link flexibility.

As soon as link or joint flexibility is considered, the order of the plant model increases, and a linear approximation is generally not sufficient. The required controller is far more difficult to design than for the case of a rigid manipulator, but several techniques to do so have been proposed in the literature. These techniques, including feedback linearization, singular perturbation, integral manifolds, and adaptive control are reviewed in Section 1.1. All of these control strategies, however, require some amount of *a priori* knowledge about the robot dynamics. For a real robot, these nonlinear dynamics are very difficult to estimate, and impossible to determine exactly. Furthermore, they may not be exactly the same for all robots of a given model, and may be environment or time dependent.

This thesis proposes a controller design which eliminates the need for *a priori* dynamics formulations by incorporating a neural network (net). A neural net can be trained to model the nonlinear dynamics of an arbitrary robot.

The next chapter reviews some neural net concepts which we will need later to select a net that is suitable for our application. In Chapters 3 and 4, we develop the general control system for use with an arbitrary nonlinear plant. A backpropagation neural net is used to model the plant's inverse dynamics. In Chapter 5, the proposed system is applied to a specific flexible-joint manipulator. Computer simulations are used to evaluate system performance with several net sizes, feedback gains and input trajectories. The system behaves as predicted, showing high conformity to the performance specification.

1.1 Current Strategies for Control of Flexible-Joint Robots

Because there exist many reliable techniques for controlling linear plants, it is tempting to apply such techniques to the control of a nonlinear plant as well. In cases where the nonlinearities cannot simply be neglected, one can attempt to linearize the plant by designing a feedback loop around it in such a way that the input/output response of the resulting system is linear. This procedure is known as feedback linearization. The computed torque method [1] for controlling rigid robots is an example of this type of linearization.

Consider the single-input n-th order nonlinear plant described by

$$\begin{aligned}
\dot{y}_1 &= y_2 \\
\dot{y}_2 &= y_3 \\
&\vdots \\
\dot{y}_{n-1} &= y_n \\
\dot{y}_n &= g_b(y_1, y_2, \ldots y_n) + g_a(y_1, y_2, \ldots y_n)\, u
\end{aligned} \tag{1.1}$$

$$\begin{aligned}
\mathit{where:}\quad y_i &\quad \mathit{are\ the\ plant's\ state\ variables}\ (i = 1,2,\ldots n) \\
g_a, g_b &\quad \mathit{are\ nonlinear\ functions\ of\ the\ state} \\
u &\quad \mathit{is\ the\ plant\ input}
\end{aligned}$$

If we apply the nonlinear control

$$u = f_a(y_1, y_2, \ldots y_n)\, v + f_b(y_1, y_2, \ldots y_n) \tag{1.2}$$

$$\begin{aligned}
\mathit{where:}\quad f_a &= 1/g_a \\
f_b &= -g_b/g_a
\end{aligned}$$

then, the resulting closed-loop system

$$\dot{y}_1 = y_2$$
$$\dot{y}_2 = y_3$$
$$\vdots$$
$$\dot{y}_{n-1} = y_n$$
$$\dot{y}_n = v$$

is linear and a simple n-tuple integrator system. This feedback-linearized system, enclosed by the gray rectangle in Figure 1.1, can now be controlled by a simple linear compensator in an outer feedback loop. It should be noted that the inner feedback loop design is totally plant-dependent in that it requires accurate *a priori* knowledge of the plant dynamics. By contrast, the outer loop design requires no such knowledge (except for the order of the system). In a sense, feedback linearization results in a new, "generic" plant which can effectively be controlled as a black box. For these reasons, the inner loop controller is sometimes referred to as the model-based portion, and the outer loop controller as the servo portion [2].

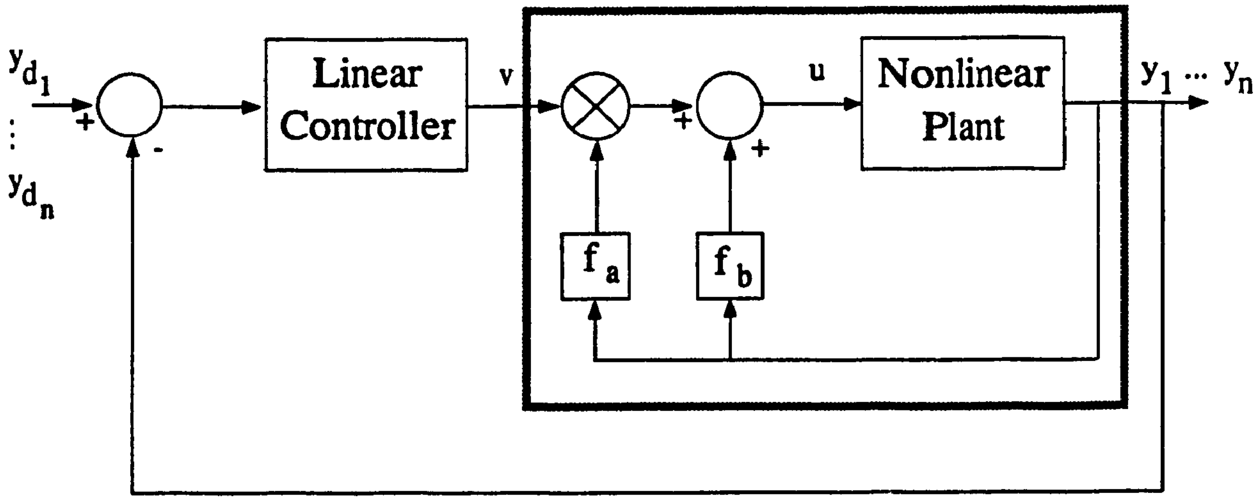


Figure 1.1: Feedback linearizing controller.

A significant difficulty with feedback linearization is that real-world plants generally do not present themselves in the form of Equation 1.1. Fortunately, it is frequently possible to obtain the required form by means of a transformation [3]. For example, given the nonlinear system

$$\begin{aligned}\dot{y}_1 &= y_1 y_2 \\ \dot{y}_2 &= \sin(y_1) + y_2 u\end{aligned}$$

we can apply the transformation

$$\begin{aligned}z_1 &= y_1 \\ z_2 &= y_1 y_2\end{aligned} \tag{1.3}$$

to obtain the form

$$\begin{aligned}\dot{z}_1 &= z_2 \\ \dot{z}_2 &= y_1\left[y_2^2 + \sin(y_1)\right] + y_1 y_2 u\end{aligned}$$

Using the inverse transformation of Equation 1.3

$$\begin{aligned}y_1 &= z_1 \\ y_2 &= z_2/z_1, \qquad z_1 \neq 0\end{aligned} \tag{1.4}$$

we arrive at the standard form

$$\begin{aligned}\dot{z}_1 &= z_2 \\ \dot{z}_2 &= z_1 \sin(z_2) + z_2^2/z_1 + z_2 u\end{aligned}$$

which can be feedback linearized by the inner loop control

$$u = \frac{1}{z_2} v - \frac{z_1^2 \sin(z_1) + z_2^2}{z_1 z_2}$$

The resulting linear system, shown inside the gray rectangle of Figure 1.2, is a double integrator. The outer loop control can be designed as before to obtain some desired response in z. The problem with such transformations is that the original plant output y is generally not linear with respect to the input v, and its response is not directly obtainable from the outer loop control law. If one is concerned with controlling the <u>original</u> <u>plant</u>

outputs (y, not z) in some specific way, as is usually the case, an additional transformation block which maps from y to z is needed at the overall system's inputs. The resulting controller is highly dependent on the accuracy of this transformation block.

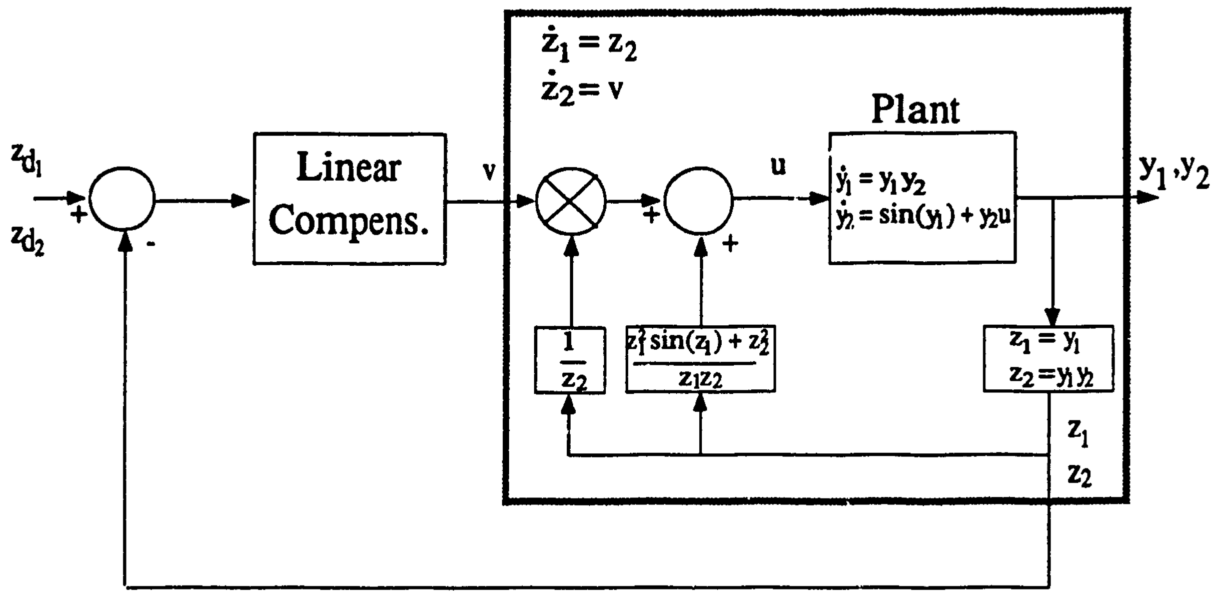


Figure 1.2: Example of a feedback linearized system with coordinate transformation.

There does not always exist a transformation which will yield a system that is linearizable by Equation 1.2. The necessary and sufficient conditions for feedback linearizability are presented in [4]. It has been shown that these conditions are quite restrictive, and preclude the linearizability of many robots that one could expect to encounter in the real world. For example, Cesareo and Marino [5] have demonstrated that a planar 2-DOF elastic joint manipulator does not satisfy these conditions, and therefore cannot be feedback linearized. Since the focus of this research is on arbitrary flexible-joint manipulators, we must reject feedback linearization as a possible approach because of its limited scope of applicability.

A more general approach can be derived from the singular perturbation formulation of the flexible-joint manipulator's equations of motion [6]. The dynamic model is decomposed into one "fast" and one "slow" subsystem. The slow variables are the joint positions and velocities, just as in a rigid model. The fast variables are the joints' internal torques and their time derivatives. This decomposition has the advantages of decoupling the fast and slow subsystems, and allowing them to be controlled separately.

Taking the singular perturbation parameter μ as the inverse of the joint stiffness, and utilizing the concept of an integral manifold, Spong *et al.* [7] have derived a reduced-order model of the flexible-joint manipulator. Their model is of the same order as the rigid manipulator, yet it incorporates the effects of elasticity. In fact, as μ tends to zero, their model reduces exactly to the rigid model. The resulting controller consists of a "rigid control" part designed for the purely rigid model, and a "corrective control" part designed to compensate for the effects of elasticity. Spong *et al.* have shown that their reduced flexible model is feedback linearizable, and that their controller is capable of accurate trajectory tracking after the decay of any fast transients. However, the integral manifold conditions may be violated during such transients, resulting in a temporary degradation of the tracking accuracy. Despite this limitation, the integral manifold approach still provides an improvement over conventional rigid control, even during transients. It also allows us to proceed when the standard feedback linearization approach is not applicable. Unfortunately, both of these techniques do share the dependence on complete *a priori* knowledge of the plant dynamics.

A number of control strategies have been developed specifically to handle some degree of plant uncertainty. Robust control involves minimizing the overall process uncertainties in terms of the system transfer function. The acceptable uncertainties are specified in terms of gain and phase variations of the plant's transfer function over a range of different frequencies. Reducing the effect of these uncertainties invariably requires a higher closed-loop gain, therefore increasing sensitivity to noise and risk of instability [8].

By contrast, adaptive control strategies deal with plant uncertainty by isolating specific plant parameters whose exact values are unknown or are expected to vary with time. The underlying assumption is that the remainder of the plant dynamics are known and fixed. Well-known adaptive systems such as MRAC (model reference adaptive control) and STR (self-tuning regulators) require, as does the Horowitz robust controller, *a priori* information about the process dynamics. It is up to the system designer to decide which parts of the plant's equations can be considered as known, and which parts are to be tuned by adaptation. The success of these adaptive control strategies depends on the accuracy with which the "known" dynamics of the plant can be represented, and on the linear parametrizability of the system. Wnen these requirements can be met, adaptive control theory provides the tools to ensure both convergence of the tracking error, and global system stability.

As an example, Slotine and Li [9] tested their adaptive control strategy on a 2–DOF rigid manipulator. Neglecting friction, the dynamics of a rigid manipulator can be written as

$$u = H(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) \tag{1.5}$$

$where:$	u	$is\ the\ n \times 1\ vector\ of\ actuator\ forces/torques$
	q	$is\ the\ n \times 1\ vector\ of\ joint\ displacements$
	$H(q)$	$is\ the\ n \times n\ inertia\ matrix$
	$C(q,\dot{q})$	$is\ the\ n \times 1\ centripetal\ and\ Coriolis\ vector$
	$g(q)$	$is\ the\ n \times 1\ gravitational\ vector$
	n	$is\ the\ number\ of\ joints$

The individual terms on the right-hand side of Equation 1.5, and consequently the dynamics as a whole can be written in terms of a linear function of a suitably chosen set of manipulator and load parameters [10]. This linear parametrizability condition allows us to rewrite the dynamics as

$$u = F(q, \dot{q}, \ddot{q})\,\phi \qquad (1.6)$$

where: ϕ *is a* $p \times 1$ *vector of selected parameters*

$F(q, \dot{q}, \ddot{q})$ *is an* $n \times p$ *matrix of known, generally nonlinear functions of* $q, \dot{q}$, *and* $\ddot{q}$

p *is the number of parameters*

Slotine and Li have developed an adaptation law to update these parameters on-line. When combined with a PD control law, this adaptation scheme guarantees global system stability and convergence of the tracking error, and does not require a reference model. Experiments with a 2–DOF manipulator demonstrated significant reductions in tracking error when compared with PD control alone, and some improvement over computed torque control. The parameter values used for the computed torque method were estimated realistically, but of course did not correspond exactly to the real values [9].

Unlike the feedback linearization method, this adaptive technique does not require *a priori* knowledge (or even estimates) of the robot parameters. However, like all other adaptive control strategies, Slotine and Li's approach depends on a knowledge of the <u>structure</u> of the dynamics. In particular, it is assumed that the dynamics are of the form of Equation 1.6, where $F(q, \dot{q}, \ddot{q})$ is fixed and known. Generally, the system will have unmodeled dynamics arising from various sources such as sensor and actuator dynamics. If the contribution of these unknown dynamics is significant, then the adaptive controller will be unable to attain the expected performance. Even if their contribution is quite small, there is an inherent danger with on-line adaptation. Because the adaptation is itself a dynamical process, it may interact with the unknown plant dynamics in such a way as to produce unwanted oscillations or even cause overall system instability.

There is no reason to believe that Slotine and Li's method could not be extended to a flexible-joint manipulator, but the resulting controller would (like feedback linearization) require the measurement of higher derivatives of joint displacements. A new method proposed by Khorasani [11] combines the Slotine and Li algorithm with the singular perturbation representation of the flexible-joint manipulator dynamic equations. This methodology results in adaptive control laws whose complexity is comparable to their rigid model counterparts.

All of the controllers discussed so far require some prior knowledge of the dynamics, with Slotine and Li's method requiring the least. An entirely different adaptive technique developed by Widrow and his colleagues [12, 13] is worthy of mention for its independence from such requirements. In this approach, the plant is treated as a "black box" containing some input/output mapping, represented by an unknown transfer function. Using adaptive signal processing techniques, a transversal filter is designed and adapted to model the plant inverse, as shown in Figure 1.3. The error signal is used to update the filter's tap weights according to some chosen adaptation rule.

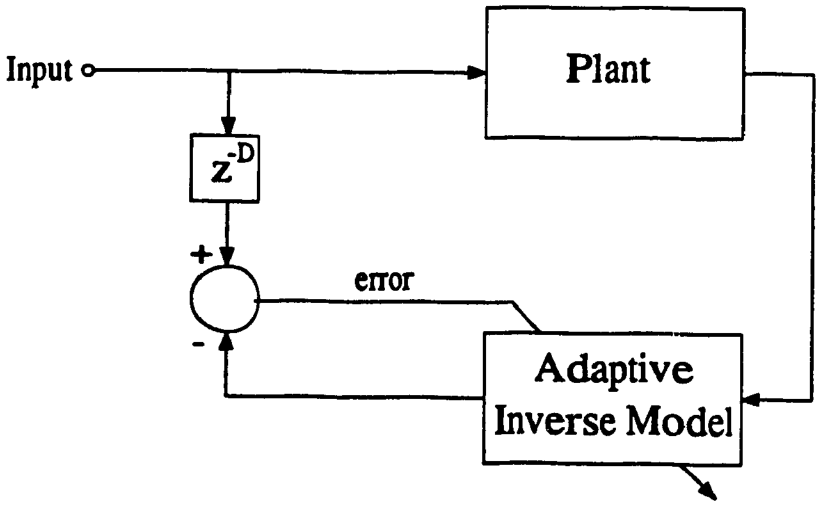


Figure 1.3: Adaptive filter for inverse plant modeling with delay [13].

The delay D is not required if the plant is minimum-phase, but by including it, we can model both minimum-phase and non-minimum-phase plants without knowing in advance with which type we are dealing. Once a sufficiently good inverse model has been obtained, the filter can be used as a feedforward compensator for controlling the plant in a feedback control system (Figure 1.4). At this point, the tap weights can be fixed or adaptation can continue on-line. If a delay ($D \neq 0$) is used in Figure 1.3, then one has to expect an on-line system delay of at least D between the reference input and the actual plant output. This may be acceptable in some, but not all cases, depending on the application.

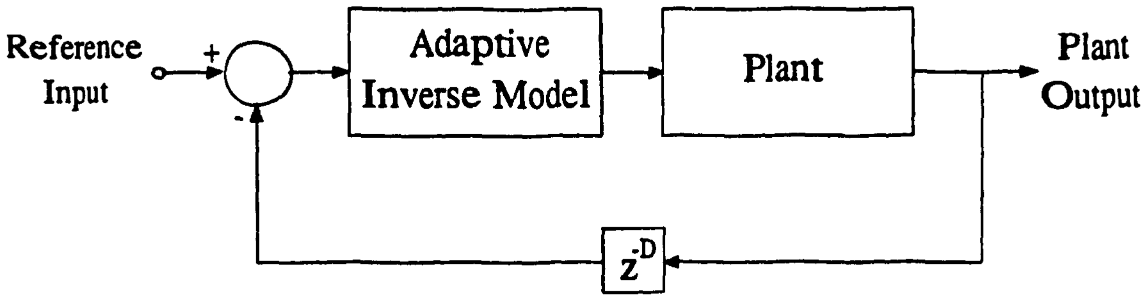


Figure 1.4: Example of an adaptive filter used to control a plant (unity feedback case).

A typical implementation of an FIR (finite impulse response) adaptive filter consists of a series of delay elements and an adaptive linear combiner as shown in Figure 1.5. An adaptation rule determines how the weights $w_0, w_1, \ldots, w_L$ will be updated, as functions of the error in the filter's response. This error is generally measured as the difference between the filter's actual output and some desired output (provided externally). For example, in Figure 1.3, the desired adaptive filter output is taken to be the plant input. A commonly-used adaptation rule is the LMS (least-mean-square) algorithm, which can also be extended to recursive or IIR (infinite impulse response) filters [13].

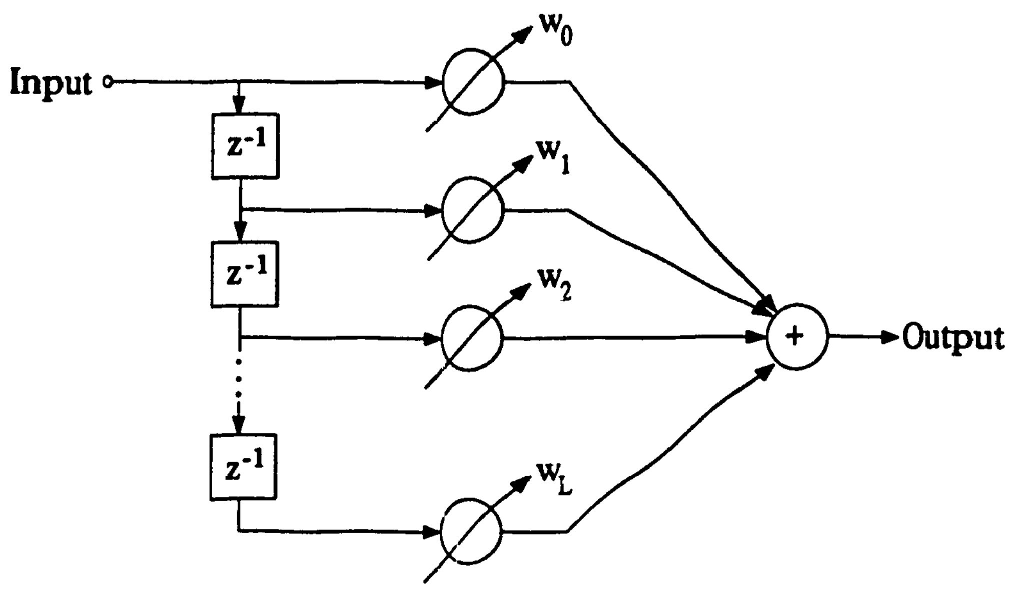


Figure 1.5: Adaptive FIR filter with L+1 weights, single input, and single output.

This approach to adaptive inverse control has the unique advantage that it requires no prior knowledge of the plant dynamics. In this sense, it is the most general technique presented so far. Although some knowledge of the plant characteristics would be helpful for choosing the length of the transversal filter (L) and the delay (D), if any, one could proceed without this knowledge by trial-and-error. Furthermore, the designer will in most cases have some idea at least of the approximate order of the plant. The main limitation is that the resulting filters are strictly linear. To generate a nonlinear model, one would have to add nonlinear transducers on each of the filter taps. Since there is currently no adaptation scheme available for modifying the nonlinearities themselves, these would have to be fixed in advance, thus requiring at least a partial knowledge of the plant dynamics.

1.2 Empirical Control Schemes

We wish to design a system which can be used to control a flexible-joint manipulator whose dynamics are unknown. Without an exact expression for the dynamics, many of the control schemes presented in Section 1.1 are rendered unusable. Adaptive control schemes allow more freedom in that only the structure of the dynamic equations must be known in advance, while the actual parameters can be generated on-line. To go one step further and eliminate our dependency even on the structure of the dynamics, we have no choice but to turn to an empirical approach.

Unfortunately, there is a very limited body of knowledge in empirically-based controller designs. The common element in such designs is that they involve some degree of learning. By learning, we refer to the controller's ability to modify its own transfer function based on some predetermined error criteria. Adaptive control is an example of on-line learning. However, learning in adaptive controllers involves the updating of only certain predefined parameters, while the nonlinearities and the structure of the equations remain fixed. The well-known neural network based technique of Kawato *et al.* [14] also models nonlinearities by preselecting and fixing them in dedicated input transducers. Neither of these approaches is completely empirical, since both make *a priori* assumptions about the system.

The required controller must be able to model the nonlinear plant dynamics by using only learning. Backpropagation neural nets and CMAC (Cerebellar Model Arithmetic Computer) networks [15, 16] are some examples of systems which have a proven ability to do so [17]. These nets use a supervised learning procedure, which means that at each step in the learning process, we must apply both a net input and an error signal to the net. The error signal is a measure of how close the net's actual output is to what it should be (desired output). This error must be computed externally to the net by some other component in the system. In many cases, there is no component in our system to provide the error signal we need. Supervised nets, therefore, are critically limited to modeling only those mappings for which the desired output is available during learning.

Of course it is possible to generate the error signal from a mathematical model [18], but such a model would require *a priori* knowledge of the dynamics which we have assumed is unavailable to us.

One mapping that can be learned by supervised nets is the inverse dynamics of a robot, as has been repeatedly demonstrated [19, 20, 21, 22]. In Chapter 3 we discuss how the error signal is derived for inverse dynamics training.

1.3 The Performance Specification

Before developing any practically useful control system, we start with some specification of the desired performance. This specification may be based on one or more of many criteria, including the system's impulse response, step response, frequency response, closed-loop pole placement, steady-state error, and transient error. A control strategy believed capable of satisfying the given specification is chosen, and the controller's parameters are then tuned (on or off-line) to actually meet the specification. For example, a rigid manipulator whose closed-loop damping ratio and natural frequency are specified could be controlled by PID feedback, where the proportional, integral and derivative terms are constants chosen off-line based on knowledge of the robot's exact dynamics.

In a real system, it may not even be possible to satisfy an overly stringent specification. In the above-mentioned PID controller, for example, it may be desirable in order to get a fast response to have the closed-loop system poles approaching $-\infty$ on the real axis of the complex plane. This will require extremely high feedback gains, which in turn is likely to cause input saturation which together with unmodeled robot dynamics and delays in the feedback loop may result in instability.

It is therefore important not to specify excessive requirements on a system, but rather to specify the minimum performance required to adequately execute the desired tasks. If the requirements cannot be satisfied by a particular controller mathematically, in simulation, or in actual operation, then one of the following actions must be taken by the system designer:

1. Choose a different control strategy.
2. Upgrade the equipment (eg. reduce delay in the feedback loop, use more precise sensors).
3. Reduce the stringency of the specification.
4. Shelve the project.

The importance of conforming to some specification must not be under-emphasized. We therefore require a control scheme with parameters which can be adjusted in some predictable manner to yield the desired performance.

1.4 Empirical Control Satisfying a Mathematical Specification

Combining the requirements discussed in the last two sections, it becomes clear that we need a controller that can empirically model the robot dynamics, while conforming to some mathematical specification for overall system response. Although these may seem to be contradictory requirements, we can in fact satisfy both by breaking up the controller into two parts. We develop this system in detail in Chapters 3 and 4. The resulting controller is similar in structure to the standard feedback linearization control strategy with the notable difference that the inverse dynamics mathematical model has been replaced by a trained neural network.

2
AN OVERVIEW OF ARTIFICIAL NEURAL NETWORKS

This chapter introduces neural networks, discussing their capabilities and limitations. The components of the network are described along with a review of available design choices, and some insight into how each choice affects the overall functionality of the net. No new material is presented here, but the remainder of the thesis makes frequent references to the issues addressed in this chapter.

2.1 Neural Network Structure

As their name implies, neural nets are loosely based on the structure of neurons in the brain. A neural net consists of a set of interconnected processing elements or nodes which are analogous to the neurons in our brains. In the typical vertebrate neuron, nerve impulses originate in the cell body, and are propagated outwards along the axon. Nerve endings called dendrites receive impulses from as many as 1000 other neurons via synaptic connections to the respective axons of those neurons. The cell body processes this information and produces an impulse which is then transmitted by a single axon to several other neurons [23]. A processing element of a neural net (Figure 2.1) is similar in that it has several inputs with varying connection strengths, a main body which processes those inputs, and a single output line which carries the resulting signal to other nodes (or neurons).

In general, the nodes are analog devices. The connection strength between the output of one node and the corresponding input of another is modifiable in both living and artificial neural systems, by the process referred to as learning.

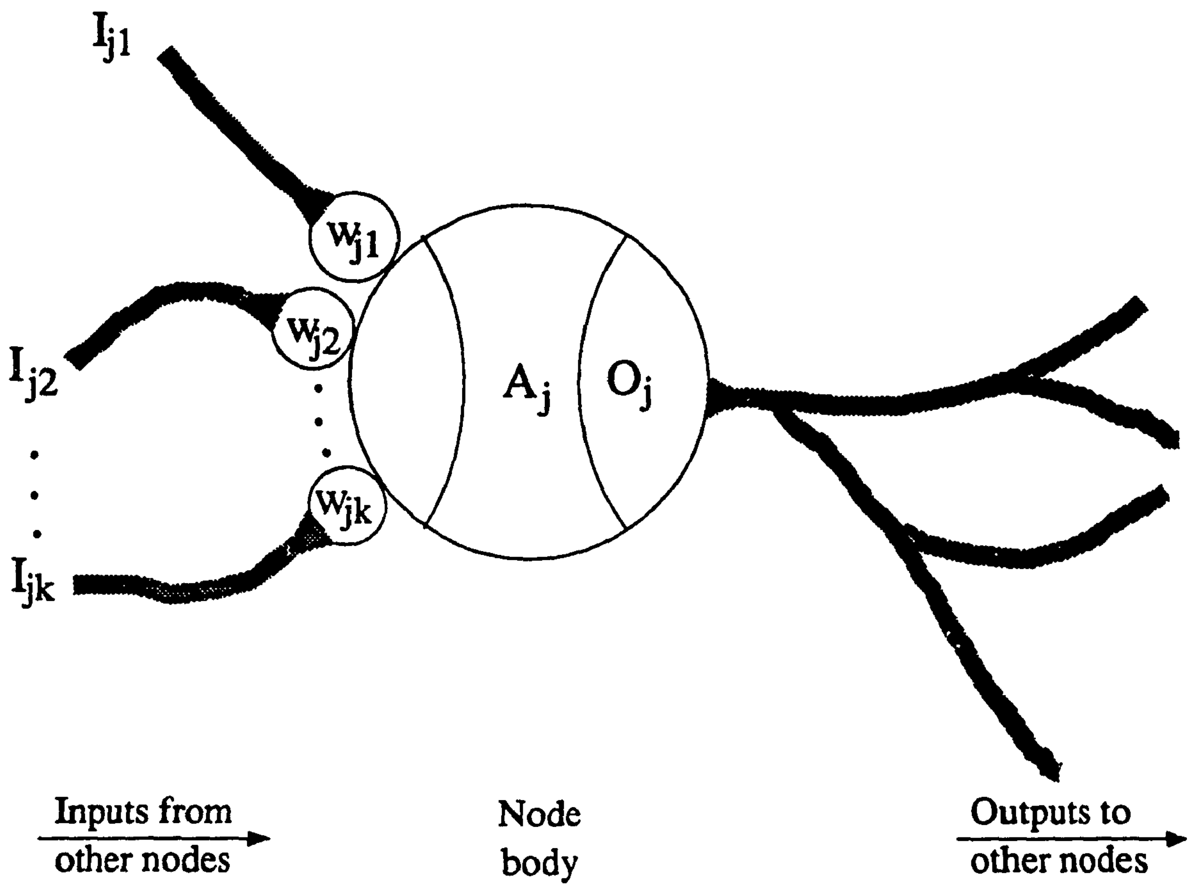


Figure 2.1: Neural Net Processing Element (Node).

There are, however, more differences than similarities between the brain and artificial neural nets. For example, the connection strengths are determined at the output branches of the living cell [24], whereas in the artificial neural nodes these weights are attributed to the inputs. Also, the internal functions of the living neurons are complex and not fully understood. They have been modeled in the artificial node by simple transfer functions such as sigmoids or sinusoids. It is therefore important not to take the analogy too far, but to be aware of it for the purpose of understanding the motivation behind neural nets and the biological terminology used to describe them.

In addition to the structure of its individual processing elements, the neural net is further characterized by the interconnection pattern between these elements. The most general pattern is the fully connected model in which each node's output is connected to every other node's input. All other connection patterns are a subset of this model which is rarely used because of its intractability. Commonly used are hierarchical patterns in which nodes are grouped into levels such that all the outputs of one level are connected only to inputs of higher levels. There exist so many different interconnection patterns that very few common features can be identified. In most cases, the interconnection pattern is quite dense when compared to parallel computers, and the number of interconnections grows disproportionately faster than the number of nodes. These are only observations however, not rules, and they do not suffice to distinguish neural nets from some parallel computer networks such as the N-cube

2.2 Learning

The appeal of neural nets can perhaps best be attributed to their unique learning capability which is not shared by traditional computer architectures. When a neural net is excited by an input signal it responds by producing some output signal. The net can be thought of as an {input : output} mapping function. The learning process involves repeated presentation of many sets of {input : desired output} data pairs until the connection strengths converge. During the learning process the connection strengths are continually changing so the same input presented at different stages of the learning may well yield different outputs. Before learning, the {input : output} mapping is generally random. Upon successful completion of learning, the connection strengths will completely describe the mapping between the input data set (domain) and output data set (range) which was used to train the net. The net thus derives a set of rules which perform the same function as the programmed algorithm running on a conventional computer.

This learning capability of neural nets makes them highly suitable for problems where some {input : output} pairs are available, but a closed-form mapping function from input to output is not known. Conventional computers, by contrast, are more suitable for problems where the inputs and the mapping function (algorithm) are fully specified. Neural nets, however, are not well suited to tasks requiring high numerical precision. For example, the multiplication of two 8-bit binary integers can be done to perfect accuracy using simple digital logic. To accomplish the same task a neural net would have to be very large and would require careful and extensive training. This is due to the essentially algorithmic nature of multiplication; knowing the algorithm, it is simple to manipulate the data, but being shown only the {input : output} data pairs, the algorithm is far from clear. Training the neural network in this case would be like expecting a child who can count but does not yet know how to multiply to complete the pattern: {2,3 : 6}, {12,4 : 48}, {1,1 : 1}, ... ,{9,7 : ?}.

It should be observed that the learning procedure is itself algorithmic in that it follows a set of rules which are predefined at the time the network is designed. However, these learning rules are very general. They depend on the physical structure of the network, but not on the problem or mapping being learned. They are more analogous to the operating system than to the applications software in a conventional computer. The same learning rules can be used with different data sets to result in completely different network behavior (I/O mappings).

It is precisely this algorithmic property of the learning rules which allows us to simulate neural networks on conventional computers. The learning algorithm tells us how the connection strengths will change each time a new {input : desired output} data pair is presented. We can thus feed training data to the simulated network to determine what its weights will converge to, but we generally cannot predict these weights *a priori.*

In summary, we wish to emphasize that neural networks are only useful for a limited (although large) class of problems: those not easily solvable by digital computers. It is precisely this capacity to complement rather than to replace traditional computing which makes neural computing such an important new technology.

2.3 The Processing Element

The typical processing element or node (Figure 2.1) is characterized by an input function and an output (transfer) function. The input function maps the inputs and their weights into the processing element's activation value. The output function maps this activation value into the processing element's output value, which is then used as an input to other processing elements.

We will use the following notation for the j-th node of a network:

I_{jk}	*is the value of the k^{th} input to node j*
w_{jk}	*is the weight on the k^{th} input line to node j*
n_j	*is the number of inputs to node j*
A_j	*is the activation of node j*
O_j	*is the output of node j*
c_1, c_2, c_3	*are constants selected by the network designer*

2.3.1 Input Functions:

By far the most commonly used input function is a single weighted summation over all the inputs:

$$A_j = \sum_{k=1}^{n_j} w_{jk} I_{jk}$$

Another function sometimes used takes the maximum (or minimum) of all the weighted inputs. Widrow's Madaline net [25] uses a majority voter input function. In yet another paradigm, a weighted multiple replaces the weighted summation, or the two can be combined in what Rumelhart calls "sigma-pi" units [26]. There are a great many variations, but in practical applications, the simple weighted sum prevails.

2.3.2 Output Functions:

The original output function used in perceptrons was the threshold function:

$$O_j = \begin{cases} 0, & A_j < c_2 \\ c_1 A_j, & A_j \geq c_2 \end{cases}$$

This is actually just an extension of the linear output function:

$$O_j = c_1 A_j$$

It is the linearity in the functions that can be blamed for the downfall of perceptrons. Just as Minsky and Papert showed that the single layer perceptron is unable to distinguish nonlinearly separable classes [27], so it can be shown that a multi-layer perceptron network is reducible to a single layer if the output functions of its nodes are linear [28].

Another commonly used function is the modified step:

$$O_j = \begin{cases} c_1, & A_j < c_3 \\ c_2, & A_j > c_3 \end{cases}$$

This function discretizes the output of the node to one of two levels. It is used in the Adaline [25] and Brain-State-in-a-Box [29] networks.

An important class of output functions consists of the semilinear mappings, in which the output of the node is a nondecreasing and differentiable function of its activation. The most common example is a sigmoid, but others such as the hyperbolic tangent can also be used. Because they are not linear, these functions can be used without redundancies in a multilayer network to overcome the perceptron limitation. Furthermore, the fact that they are nondecreasing and differentiable means that the generalized delta learning rule [30, 26] can be used to train such a multilayer network. The result is the backpropagation network which is capable of classifying nonlinearly separable data.

2.4 The Topology

The most general network topology from which all others are derived is the fully-connected model. In such a network every node's output is connected to one input of each of the other nodes, including a feedback connection to itself [31]. If there are N nodes, then each node must have N inputs and there are in total

$$2 * \binom{N}{2} + N = N(N-1) + N = N^2 \ \textit{connections.}$$

Thus, for a net with 100 nodes, we would need 10,000 connections. Such a ratio is very difficult to implement in hardware. In fact most commercial neural processor boards offer ratios ranging from 10 to 25 connections per node. Furthermore, the behavior of such a system is very difficult to analyze and relaxation may be very slow, to the point of making the net unsuitable for simulation.

To make their nets more tractable and the structures more meaningful, most researchers have adopted a hierarchical topology as shown in Figure 2.2. Nodes are grouped into layers and the layers are numbered starting with the input layer. The input layer usually acts only as a buffer and is omitted from some networks, however in the following discussion, we assume its presence.

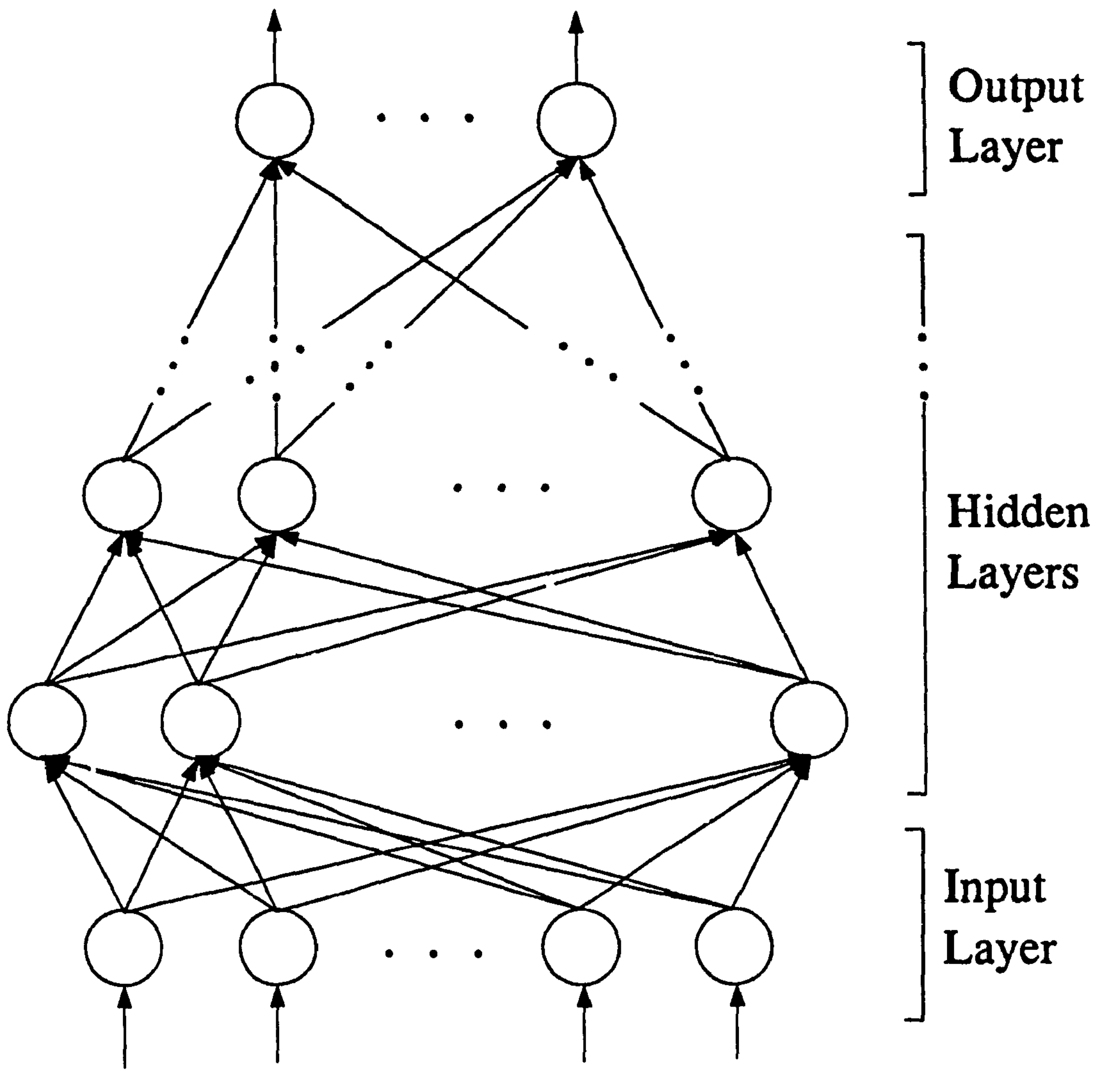


Figure 2.2: Hierarchical Network Topology.

2.4.1 Number of Layers:

One of the first steps in designing a neural net is deciding how many layers are required. If all nodes have nonlinear output functions, then increasing the number of layers generally enables the net to learn more complex patterns. To illustrate this, Lippmann [28] uses the example of a 2-dimensional (2 inputs) pattern classifier with step output functions on all nodes. A single-layer version of this network (ie. output layer and input buffer but no hidden layers) forms half-plane decision regions and is therefore only able to distinguish linearly separable classes. The two-layer net (ie. one hidden layer) forms a convex decision region in the input space. The three-layer net can combine several convex decision regions to form any arbitrary shapes in the input space. The network designer attempts to select the minimum number of layers required for his problem. If he is uncertain about his data distribution, as is usually the case, he may have to experiment with different numbers of layers. It should be noted that in Lippmann's example, more than three layers are never required, however in some cases, adding more layers may actually reduce the total number of nodes or improve network convergence.

In most neural nets, all the nodes within a given layer are of the same ype. They have the same input and output functions, and they follow the same learning rule. Different layers, however, may be composed of different nodes. For example, Widrow's Madaline network [25] with one hidden layer has simple summation input functions in the first two layers, and a majority input function in the last (output) layer. His hidden layer follows the Widrow-Hoff [25] learning rule, whereas the input and output layers perform no learning at all.

2.4.2 Number of Nodes:

The number of nodes in the input buffer layer is equal to the number of inputs, and the number of nodes in the output layer is equal to the number of outputs. Choosing a "good" number of nodes for the hidden layer(s) can however be difficult. Lippmann describes the relationship between the number of nodes and the richness of the training data: "The number of nodes must be large enough to form a decision region that is as complex as is required by a given problem. It must not, however, be so large that the many weights required cannot be reliably estimated from the available training data" [28]. As was the case with choosing the number of layers, nothing more precise can be said about the number of nodes without knowing how the data is distributed. Experimentation is the only way by which to proceed.

2.4.3 Direction of Information Flow:

A network can be classified as either of feedforward type or of feedback (bi-directional) type. In both cases, the external connections are the same: data is entered at the input nodes and is generated at the output nodes. Internally, however, the two types differ. In a feedforward net, each node's inputs are connected only to the outputs of nodes on lower levels. An excitation applied to the input will propagate forward through the layers until it reaches the output. This system has a constant and relatively short response time to a step input. Any given constant input will always produce the same steady-state output, regardless of previous inputs. The perceptron and the Adaline are feedforward nets.

In a feedback network, some nodes receive input from nodes in both lower and higher layers. Nodes in one layer can also receive input from each other. To avoid endless oscillations or divergence in such systems, the nodes have clipped nonlinear output functions. When an input is applied, the feedback connections cause some nodes to become more active and others less active. The system is said to have converged or "relaxed" once every node has reached its upper or lower output limit. In a discretely sampled net (or simulation), this process typically requires several time samples (or iterations).

Because the node outputs converge to one of two states, feedback nets are used with binary input and output data. Feedforward nets are suitable for both binary and continuous data. Another difference is that the next state of a feedback net depends not only on the current inputs but also on the previous state. A well-known feedback network is Anderson's Brain-State-in-a-Box (BSB) [29].

2.5 The Learning Rule

Much like a Von Neumann computer without software, an untrained neural net can perform no useful function. It is the learning (or training) process which gives the network all its "knowledge". Since the information in a net is stored within the internodal connection strengths or weights, the learning process consists of correctly setting these weights. At each learning iteration, an input and a desired output are presented to the net which modifies its weights so that the net's actual output approaches the given desired output. Starting with the same untrained net, different learning sessions can result in trained networks having dramatically differing functionalities. The three factors involved in training are:

1. the set of training data used
2. the order in which data from this set are presented, and the number of times they are repeated
3. the learning rule

The training data consists of {input : desired output} pairs chosen to be representative of the expected input domain. Often, the training points are evenly distributed throughout the input space. A common variation is to divide the input space into regions with the number of training points in each region proportional to the probability of an input occurring in that region.

The order of presentation is usually random (or pseudo-random) to inhibit the formation of artificial local minima in the activation space. The training set is presented repeatedly as many as hundreds or even thousands of times. To determine how many times is enough, the network output can be monitored, and learning is terminated once the net's "hit ratio" reaches some desired threshold. The hit ratio is some measure of the net's performance, such as the frequency of correct classification (net output matches desired output). In exceptional cases, learning can be terminated after less than five repetitions of the training set. Although it is most common to first train the network, and then to employ it with the weights fixed, it may be desirable to continue training while the net is in service. For example, if the net is used within some larger system which can measure output error, this error can be fed back to the network as a training signal. Such on-line training systems have been proposed for adaptive robot control [29].

The learning rule determines how the weights at each level change in response to one training data point. In supervised learning, the weight changes are a function of the difference between the network's actual response to an input and some desired response. Generally, the greater this difference, the more the weights change. In unsupervised learning, the desired output is not given, but is assumed to be the same as the training input. The remainder of this section presents some commonly used learning rules. In all cases, supervised learning is assumed, but the rules can be extended to unsupervised learning by setting the desired output equal to the training input.

The following additional notation will be used to describe the effect of a single learning iteration on the weights of the j-th node:

D_j	*is the desired value of the node's output*
w_{jk}	*is the current weight on the* k^{th} *input line*
Δw_{jk}	*is the change in* w_{jk} *to result from the current learning iteration*
n_j	*is the number of inputs to node* j, *and* $1 \le k \le n_j$

2.5.1 Hebbian Learning

Hebb's original learning rule was based on the principle of increasing a synaptic strength whenever the corresponding actual input and the desired output are simultaneously active:

$$\Delta w_{jk} = \begin{cases} c_1, & if\ D_j > c_2\ and\ I_{jk} > c_2 \\ 0, & otherwise \end{cases}$$

$$\begin{array}{lll} where: & c_1 > 0 & is\ the\ learning\ rate \\ & c_2 & is\ a\ threshold\ above\ which\ a\ node\ is \\ & & considered\ active \end{array}$$

Although this rule is important historically, it has the major disadvantage that it has no mechanism for reducing the weights. Several variations of Hebb's original rule offer this facility including Hopfield's modification [32]:

$$\Delta w_{jk} = \begin{cases} c_1, & if\ (D_j > c_2\ and\ I_{jk} > c_2)\ or\ (D_j < c_2\ and\ I_{jk} < c_2) \\ -c_1, & otherwise \end{cases}$$

The effect of this rule is to increase the k-th weight whenever the k-th input is the same as the desired output (ie. both are active or both are inactive), and to decrease the k-th weight otherwise. Hopfield's model used the step output function:

$$O_j = \begin{cases} 0: & inactive \\ 1: & active \end{cases}$$

with $c_2 = 0.5$. The result was a discrete binary model, although in principle this learning rule could be used with continuous models.

2.5.2 Perceptron Learning

Hebbian learning and its variations fail to take into account the actual output of the node whose input weights are being trained. Thus, the weights may be modified even if the node's actual output matches its desired output. By contrast the perceptron learning algorithm [27] treats the difference between actual and desired outputs as an error term and modifies the weights by an amount proportional to this error:

$$\Delta w_{jk} = \begin{cases} c_1 * (D_j - O_j)/n_j, & if\ I_{jk} > 0,\ D_j > 0\ and\ O_j \le 0 \\ -c_1 * (D_j - O_j)/n_j, & if\ I_{jk} > 0,\ D_j \le 0\ and\ O_j > 0 \\ 0, & otherwise \end{cases}$$

Effectively, if the output should be active but is not, the weight is increased on all active input lines. If the output should not be active but is, the weight is decreased on all active input lines. The weight changes become increasingly smaller as the weights converge to their final values. Furthermore, the convergence of perceptron learning has been proven for linearly separable input sets [27]. Although this learning scheme was initially used with linear node output functions, there is no reason why it could not be extended to nonlinear or semilinear output functions.

2.5.3 Widrow-Hoff Rule

Whereas Hebb's rule depends on the actual input and the desired output, and the perceptron rule depends on the actual input, the actual output and the desired output, the Widrow-Hoff rule combines dependencies on the actual input, the weighted sum of all the inputs, and the desired output. Developed by Widrow and Hoff for their Adaline network [25, 12], this rule can be summarized as

$$\Delta w_{jk} = c_1 * (D_j - A_j) * I_{jk} / n_j$$

The Adaline output function is the nonlinear step with the two possible values -1 and $+1$. By using the node's activation A_j instead of the actual output O_j, we allow for a variable amount of weight change (like the perceptron rule), while maintaining a nonlinear output function. The I_j term only affects the sign of the weight change.

A variation which uses the actual output O_j instead of the activation A_j is employed in Anderson's BSB model [29]. This can be done because unlike the Adaline, the BSB's middle layer uses linear output functions.

2.5.4 Delta Rule and Backpropagation

The Hebb, perceptron and Widrow-Hoff learning rules all require *a priori* knowledge of the desired outputs for all the nodes being trained. In general, however, when training a network, we only know the external inputs and outputs of the net. The desired outputs of hidden nodes within the network are not known. As a result, only the output layer (whose desired outputs are known) can be trained by the preceding learning rules. There can be several layers in the net, but all except the output layer will have fixed weights set by the network designer. These learning rules therefore limit the capabilities of all nets to those of a single-layer net.

The generalized delta rule [30, 26] is very similar in form to the Widrow-Hoff rule except that all signals are now assumed continuous:

$$\Delta w_{jk} = c_1 * \delta_j * I_{jk} \tag{2.1}$$

where δ_j is the error signal, and c_1 is referred to as the learning rate constant. For each node j in the output layer, the error signal is given by

$$\delta_j = (D_j - O_j) * f_j'(A_j) \tag{2.2}$$

where $f_j'(A_j)$ is the derivative of the j-th node's output function

$$O_j = f_j(A_j)$$

For each node j not in the output layer, the error signal is given by:

$$\delta_j = f_j'(A_j) * \sum_{q \in Q} (\delta_q * w_{qj}) \tag{2.3}$$

where the summation is taken over all nodes in the layer directly above node j, and w_{qj} is the weight on the connection from node j to node q.

The backpropagation learning procedure applies the delta rule to multilayered feedforward nets. For each (input : desired output) data pair presented to the net, the following steps are executed:

1. The inputs are applied and fed forward through the network, until all node outputs stabilize.

2. Starting with the output layer, the error signals are computed for each node and the input weights to all nodes in that layer are updated as explained above.
3. Step 2 is repeated layer by layer going downward until the lowest hidden layer is reached.

The output function must be continuous and differentiable. It should also be noted that if the output function is linear, then its derivative is a constant and the delta rule for the output layer reduces to the Widrow-Hoff rule. The semilinear sigmoid function is most often used.

2.5.5 Other Schemes

There are many other learning schemes and variations of the above rules described in the literature. One such family of learning rules is due to Kohonen [33] and involves changing the weights so that they correspond to an average of all the input vectors presented during learning. This rule is used in the hidden layer of Hecht-Nielsen's counterpropagation network [34].

Another approach has been to describe the state of each node by a probability density function. Anderson and Abrahams, for example, proposed a Bayesian probability network [35] in which all inputs and outputs were given as probabilities, and the state of the overall system was represented as an energy function analogous to Hopfield's energy function [32].

A difficulty often encountered when using neural nets for a practical problem is the appearance of spurious states as learning progresses. Several methods have been proposed to reduce this effect. In Hopfield nets, a procedure called "unlearning" [36] has been used to cancel these spurious states. The trained network is presented with random inputs whose resulting outputs are used to modify the weights in a direction opposite to the way they would be modified during learning. By repeating this for random inputs, the spurious states are "unlearned" first because they are weaker, but unlearning must be stopped before the previously learned states are also unlearned. This procedure has been described as functionally analogous to dream sleep.

Another procedure called simulated annealing [37, 26] is used to escape local minima in the energy function of the network. The name originated from the annealing process used to cool molten metals to their lowest (most stable) energy state. In simulated annealing a "temperature" coefficient is slowly reduced while the units in the network are allowed to change state according to a probability function. When the temperature is high the states change dramatically with relatively little dependence on the system energy. At low temperatures, the states have a strong tendency to change so as to reduce the overall energy, but the changes themselves are very slight. Reducing the temperature slowly therefore allows the system to initially escape local energy minima, and to eventually settle at the global minimum (most of the time). The Boltzmann machine [37] combines a probabilistic decision function, a modified version of unlearning, and simulated annealing into a single network training procedure.

2.6 Supervised Learning & Hetero-Associativity

One of the greatest difficulties encountered during the course of this research has been in obtaining a suitable training signal for the net. The following question has repeatedly arisen: "Is it not possible to somehow learn the inverse dynamics using an unsupervised learning scheme?" Unsupervised learning has the advantage of not requiring an external teaching signal, whereas supervised learning by definition does require such a signal. As shown in Figure 2.3(a) this "supervisory" signal must be in the same units as the network output. It is usually a desired net output which is compared to the actual net output at each learning step. The resultant error signal, also in net output units, is used internally by the net to update its weights. In some cases the error signal itself may be directly available as shown in Figure 2.3(b). Both of these configurations are referred to as "hetero-associative" because they map (associate) a vector from the input space into another vector from a different output space. In general, the two spaces are of different dimensions. Hetero-associative nets are therefore used for modeling the input/output relations of systems.

By contrast, auto-associative nets (Figure 2.4) have input and output spaces of the same dimension. There is a limited number of valid output vectors for which the network is trained. Any given input vector maps to the closest[1] valid output vector. This net is especially useful in the presence of noise. If a noisy or incomplete version of a valid input pattern is presented, the network will output the original pattern, noise-free and complete. Auto-associative nets are mostly used for pattern classification and noise rejection. They are trained by repeatedly applying vectors from the valid output set to the net's input. The learning is unsupervised because no external teaching signal is required. However, to adjust their internal weights during learning, all networks require some error signal. Auto-associative networks derive this signal internally by comparing their output with their input which also happens to be the desired output during training.

Auto-associative nets can therefore be thought of as a subset of hetero-associative nets, wherein the assumption of identical input and output space has been made, and consequently the error signal required for learning can be obtained internally. Auto-associative nets are not suited to our application of robot dynamics modeling, so we must use a hetero-associative net with supervised learning. Therefore, in our control system, we will be able to use a neural net to model only those relations for which a teaching signal is available.

[1] "Closeness" is usually measured in terms of mean-squared error but can be based on other criteria as well.

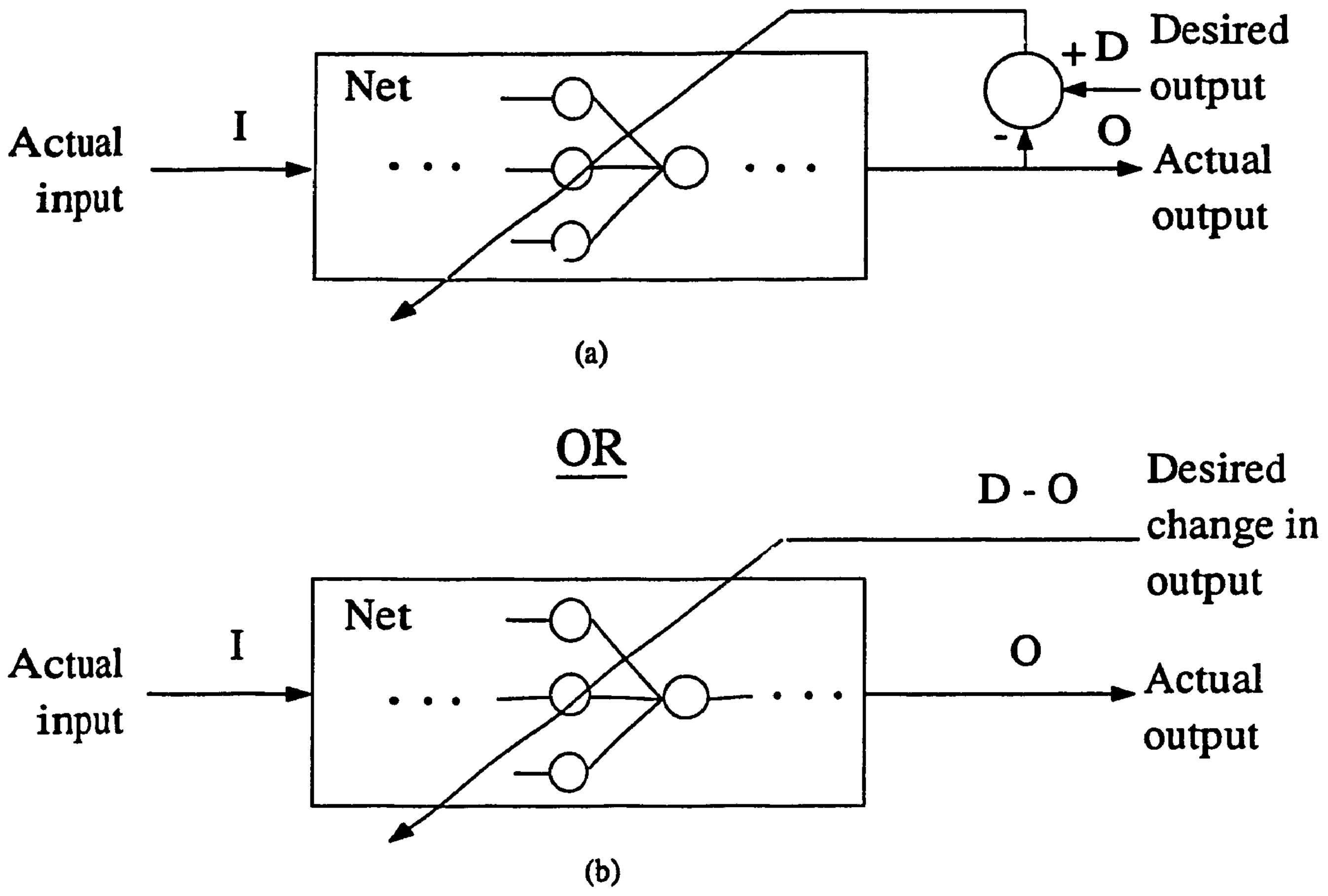


Figure 2.3: Hetero-associative network learning requires an external supervisory signal.

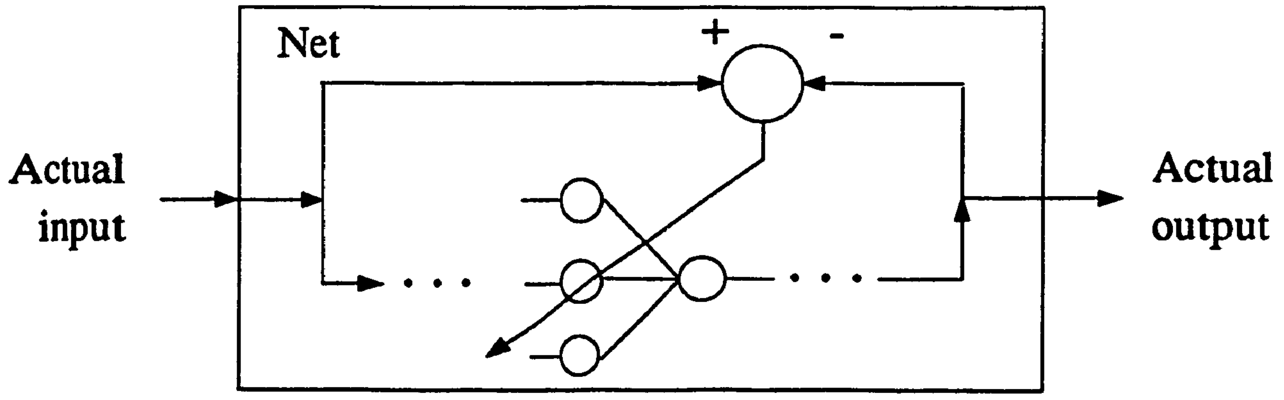


Figure 2.4: Auto-associative network learning requires no external supervising signal.

2.7 Simulation Control Schemes

Since most neural nets described in the literature are not implemented in analog hardware, but are simulated on digital computers, the network designer must take into consideration the effects of the resulting discretization.

2.7.1 Synchronous vs. Asynchronous Update

Synchronous update assumes that the outputs of all nodes change either simultaneously or in some predetermined order. For example, in simulating a feedforward network, the software would first compute the outputs of the nodes in the input layer, one at a time. Then the outputs of that layer would be used as inputs to compute the outputs of the nodes in the layer above, and so on until the output layer. A real neural network generally does not have a global synchronization strategy. Each node updates its output continuously based on its inputs. Furthermore, different nodes can have different response rates. Such an asynchronous update scheme can be simulated using a pseudo-random control strategy on a digital computer, but this is more difficult both to implement and to analyze. In some cases, such as that of the single-layer perceptron, both update strategies will yield the same final result, although the synchronous update will be faster. However in cases involving feedback, different update strategies will result in different paths along the system energy function, with possibly different results. In such cases, the designer must choose the strategy which better suits his assumptions about the system.

2.7.2 Relaxation Time

In feedback nets such as Bidirectional Associative Memory [38], or in feedforward nets with probabilistic output functions, a period of time is required for the system to stabilize after an input is applied. This process, known as relaxation, corresponds to the network converging to an energy minimum. Simulation involves repeatedly updating the node outputs until some stability criterion is satisfied. The criterion can be a certain number of update iterations, or some error margin. The choice of this stability criterion can have a major effect on the results obtained from the net.

2.8 Conclusions

As we have shown, there is presently no such thing as a "generic neural network". A great many options must be considered in selecting a specific neural net for a particular application. In addition to the more common, frequently used network paradigms summarized in this chapter, the designer is free to create hybrid paradigms by combining characteristics from various standard paradigms. One can also introduce entirely new strategies as one considers appropriate to a specific application.

Beyond the distributed structure and the ability to learn, there are currently no fixed rules constraining the architecture and functionality of artificial neural nets. There is also extremely little theory and analytical work available to guide the designer in making the best choices. Researchers working in the area often comment that the network design process is still very heuristic and generally involves some degree of trial and error. However, we have identified the following areas which must all be considered when setting up a net:

1. Associativity (hetero- or auto-)
2. Resolution (continuous, 2-state, tri-state, or discrete multiple states)
3. Learning Rule
4. Number of layers
5. Number of nodes per layer
6. Direction of flow (feedforward or feedback)
7. Node Input Function
8. Node Output Function
9. Simulation Control Scheme

10. Amount of theoretical knowledge available on the paradigm selected (an existing proof of convergence or proof of stability may be a strong incentive to use a particular net)

In each of the areas, a different approach can be taken for different layers of the net. For example, Hecht-Nielsen's counterpropagation net has a middle Kohonen layer within which the nodes compete on a "winner takes all" basis. The output layer by contrast is the Grossberg outstar with a simple weighted sum transfer function in the nodes [39]. Even in relatively homogeneous nets such as backpropagation, it is common to have a different learning rate in each layer.

In Chapter 3, we consider each of the above-listed areas in selecting the net which is most suitable for our particular application of nonlinear dynamics modeling.

3
TEACHING MANIPULATOR INVERSE DYNAMICS TO A NEURAL NETWORK

Several research groups have successfully trained neural nets to model the inverse dynamics of rigid link/joint manipulators. There is a wide variance in the approaches used and in the assumptions made about the plant. We review some of these techniques in Section 3.1, and discuss their lack of compatibility with our own design objectives. The remainder of this chapter develops a neural net along with a training procedure to meet these objectives.

3.1 Approaches to Inverse Dynamics Training

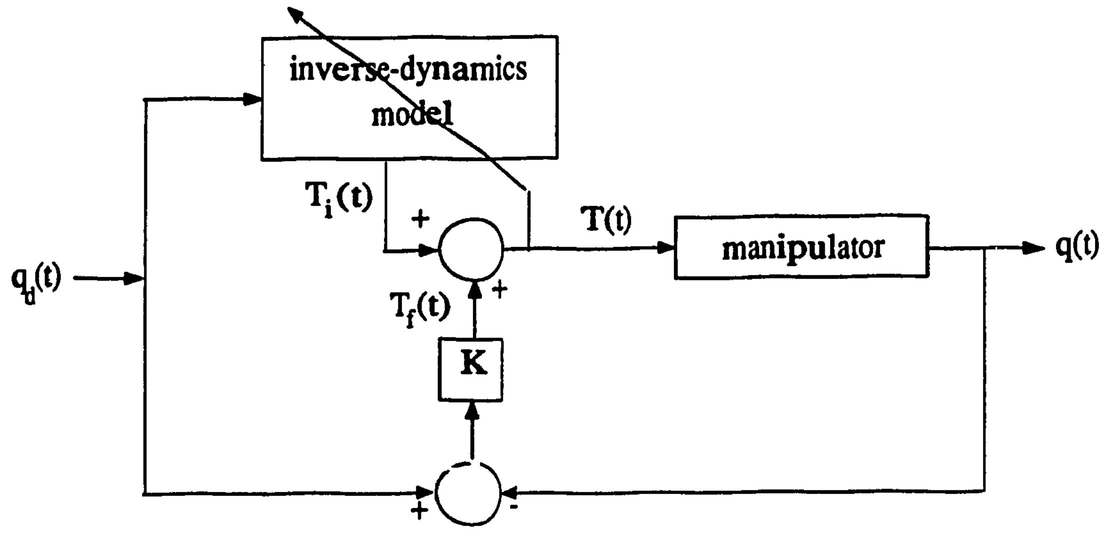
In designing a neural-network based robot controller, one faces two key questions:

1. What network configuration is capable of providing the modeling capabilities required?
2. From where will the training signal (desired net output) be obtained?

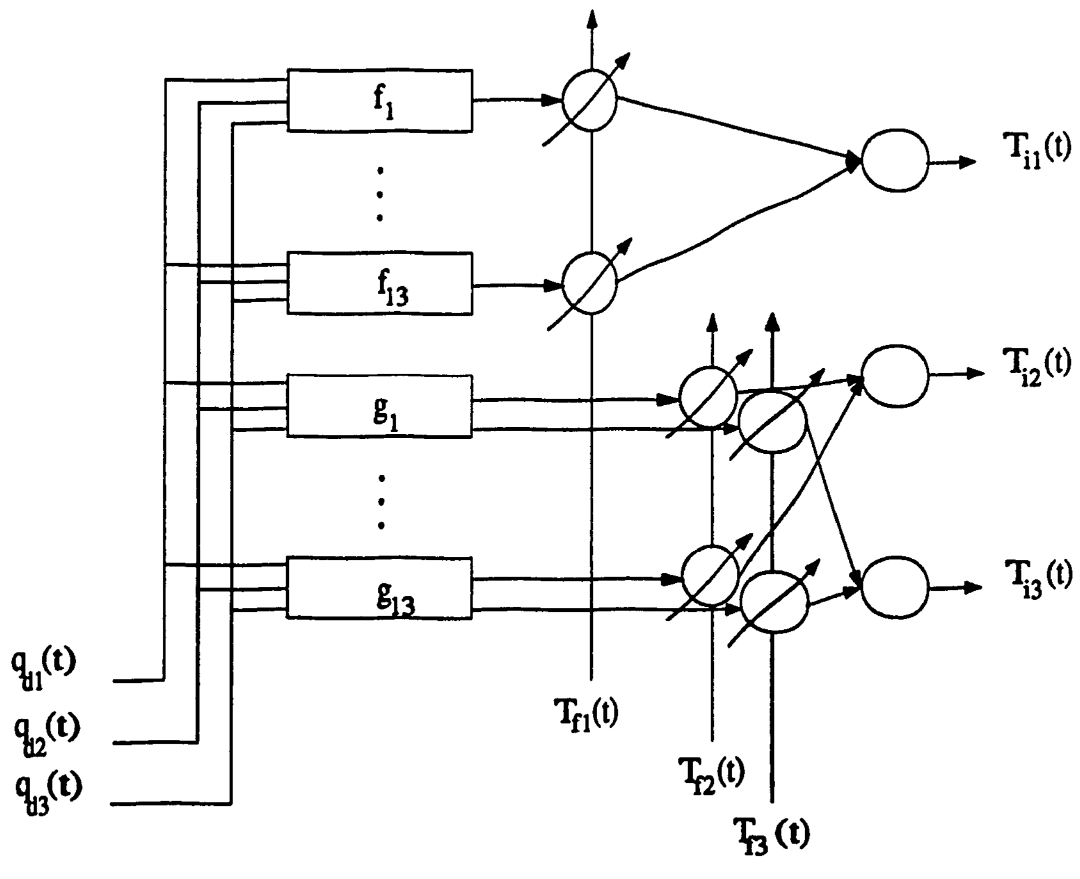
This section presents the work of several researchers who have used neural nets to model inverse robot dynamics. Their approaches will be compared and contrasted on the basis of how they resolved these two questions.

Among the pioneers and frequently quoted researchers in the area are Kawato *et al.* [14, 20]. They have trained a neural net to model the inverse dynamics of a 3 degree-of-freedom rigid manipulator. Their net is used as a feedforward controller and is trained on-line as shown in Figure 3.1(a). A fixed-gain PD feedback controller is used to compute the torque error from position and velocity errors and feed it to the neural net as a training signal. The PD gains are calculated based on a linearization of the manipulator dynamics. The net itself is a single layer linear perceptron-type network as shown in Figure 3.1(b).

Functions $f_1 - f_{13}$ and $g_1 - g_{13}$ are nonlinear "subsystems" consisting of sines, cosines, multiplications and differentiations. The functions are selected *a priori* so that a linear weighted summation of them would yield the manipulator's exact dynamical equations. There is no learning within these nonlinear subsystems but only in the weights of the linear summation. The net was trained by repeated presentations of one trajectory, and demonstrated quite accurate control performance when used for that particular trajectory. Other trajectories required re-training. The approach of Kawato *et al.* requires *a priori* knowledge of the nonlinearities in the manipulator dynamics.



(a)



(b)

Figure 3.1: Kawato's Inverse Dynamics Controller. (a). block diagram of simulated neural net and manipulator. (b). internal structure of neural net [20].

Selinsky and Guez [18] proposed a similar structure for their rigid robot neuro-controller. As with Kawato *et al.*, a single layer perceptron-type net performs a linear weighted summation on a group of nonlinear functions f_{ij}. Selinsky and Guez, however, have implemented each of these nonlinear functions as an independent backpropagation neural net.

The closed form dynamics of a rigid-link/joint manipulator can be expressed as:

$$u = F(q, \dot{q}, \ddot{q})\ \phi$$

$where:$	u	$is\ the\ n \times 1\ vector\ of\ actuator\ forces/torques$
	q	$is\ the\ n \times 1\ vector\ of\ joint\ displacements$
	$F(q, \dot{q}, \ddot{q})$	$is\ an\ n \times p\ matrix\ of\ \underline{known}\ functions$
	ϕ	$is\ a\ p \times 1\ vector\ of\ weighting\ constants$
	n	$is\ the\ number\ of\ joints$
	p	$is\ the\ number\ of\ weights\ in\ the\ net's\ output\ layer$

There are generally several equivalent systems which satisfy the above equation. The approach of Selinsky and Guez is to select one such parameterization based on *a priori* knowledge of the structure of the dynamics equations. The exact values of certain constant parameters such as link lengths and masses, however, need not be known in advance. The $(n \times p)$ neural subsystems are trained individually off-line to model the component functions $f_{ij}(q, \dot{q}, \ddot{q})$ of the matrix $F(q, \dot{q}, \ddot{q})$. Once trained, the subsystems are combined to form the neurocontroller wherein only the parameters ϕ are then learned on-line.

Whereas Kawato *et al.* implements the nonlinearities as mathematical functions, Selinsky implements them as neural nets which are trained to model those same functions. The effective difference is that the implementation of Kawato *et al.* is more precise. The nonlinearities of Selinsky and Guez are not tuned or re-trained on-line any more than those of Kawato *et al.* Selinsky and Guez claim that the advantage of their method lies in the greater fault-tolerance of their distributed implementation.

There is certainly an overlapping region between the fields of neural control and adaptive control. Just as the single-layer perceptron net is nothing more than Widrow's adaptive linear combiner (Figure 1.5), so also, the neurocontroller of Selinsky and Guez has exactly the same macrostructure and parameter updating rules as the adaptive control scheme of Slotine and Li [9] (Section 1.1). On occasion, the question of whether a particular strategy is adaptive or neural has been the source of heated discussion. However, for the purpose of this thesis, we can accept these controversial schemes as neural candidates because of their ability to adapt or "learn." Without worrying about taxonomy, we can see that the neural scheme of Kawato *et al.* and that of Selinsky and Guez share with the adaptive scheme of Slotine and Li a dependence on accurate prior knowledge of the structure of the manipulator dynamics. We, however, are seeking a more general solution to the control problem at hand.

The inverse manipulator dynamics represent a $2N$-th order system for N-link rigid robots (higher order for flexible robots). By contrast, the nets discussed so far perform strictly feedforward zero-order (static) mappings from inputs to outputs. The derivatives $(\dot{q}, \ddot{q})$ of the input q have to be generated externally and supplied to the net as additional inputs. An alternate approach would be to use recurrent nets such as those proposed by Hopfield [32]. These dynamic nets incorporate internal feedback via integrators (time delays in the discrete case). The result is a higher order system which can be represented by a set of differential equations of the form:

$$\dot{x} = -\alpha x + N(x) + I$$

where : $x(t) \in R^n$ *is the state of the net at time t*

$N(x)$ *is the mapping function of the internal static subnetwork*

$I \in R^n$ *is the net input*

$-\alpha$ *is the pole*

By using a multiple layer nonlinear neural net for the mapping function $N(x)$, it has been argued that the resulting class of generalized neural nets is "adequate to deal with a large class of problems in nonlinear systems theory" [40]. Unlike the standard feedforward net which can only model static mappings, these generalized nets can potentially model a dynamical system of arbitrarily high order, without requiring any external computation of input (and output) derivatives. Therefore, the inverse dynamics of a single-joint manipulator can theoretically be modeled by a single input (joint position), single output (torque) generalized neural net.

Aside from the fact that very little is known about such nets, there are some fundamental dangers with this approach. Because this net internally performs integration on the weighted input signals, even small modeling errors or biases can become very large with time. To prevent this, the net may require an additional feedback loop or some external "synchronization" signal. A still more severe problem occurs if the plant is non-minimum-phase, that is, in the case of a continuous linear time-invariant system it has zeros in the right-half s-plane (or outside the unit circle in the z-plane for a discrete linear time-invariant system). The dynamical net (plant's inverse dynamics) would then have poles in the right-half s-plane (outside the unit circle in the z-plane), and would therefore be open-loop unstable. Even if the overall closed-loop system is stable, the neural net itself could saturate, causing severe clipping errors.

One way of dealing with this problem is to introduce a delay into the system, as proposed by Widrow and Stearns [13]. The corresponding training configuration is shown for a discrete system in Figure 1.3, where the adaptive inverse model can be replaced be a recurrent net. By choosing a sufficiently large delay D, the delayed inverse model can be stabilized, while preserving its amplitude response. Unfortunately, this method generally causes significant changes in the phase response (and transient response). Although this solution may be acceptable for certain filtering applications, it certainly does not lend itself to real-time robot control. Widrow and Stearns have suggested methods to restore even the phase response, including the addition of a phase compensator with variable delay, but any such add-ons can at best degrade the overall system response.

In any case, some additional delay in the response is necessary to ensure stability of the inverse of a non-minimum-phase system. Even if we were to accept a small known delay D in the controller, there is no reliable method for selecting D for a plant whose dynamics are unknown. Choosing a very large D just to be safe would likely violate our system performance requirements. Because we assume no prior knowledge of the plant's dynamics, we cannot even be sure whether or not they are minimum-phase. The risk of instability must therefore be assumed in all cases, making the dynamical net an unreliable solution for the purpose of this work, but certainly one that merits future investigation.

Finally, a neural net paradigm worthy of mention for its relatively short training times is the counterpropagation net. It is typically composed of three layers. The first hidden layer normalizes the input vectors onto the surface of a unit sphere. The second hidden layer is a Kohonen layer within which the nodes compete among themselves such that there is only one "winner" for each input vector presented. Thus, only one node in this layer is active at any given instant. The final output layer is the Grossberg Outstar. It maps each of the finite number of possible outcomes of the Kohonen layer into a point in the output space [39].

Hecht-Nielsen claims that the counterpropagation net can usually be trained one to two orders of magnitude faster than the backpropagation net [39]. The shorter training time has been confirmed in simulations of robot kinematics [41]. However, in both [39] and [41], it has been concluded that the counterpropagation net results in significantly greater error when modeling a continuous system. This is due to the discretization in the Kohonen layer. In fact, the number of levels of discretization in the trained counterpropagation net is bounded from above by the number of nodes in the Kohonen layer. Of course the discretization error can be reduced by increasing the number of nodes, at the expense of the required training time. However, this approach seems counter-intuitive since it involves using a discrete network to model a continuous function.

3.2 Selecting a Network Paradigm

Referring to the criteria for network selection listed in Section 2.8, we must now select a suitable neural net for modeling robot inverse dynamics. Robot inverse dynamics map joint positions q (and possible higher derivatives of positions $\dot{q}, \ddot{q}, \ldots$) into a driving torque u. This is clearly a mapping from one space into another entirely different space, so the neural net will be hetero-associative. From the discussion in Chapter 2, it follows that we will use a supervised learning strategy and will consequently require a training signal from outside the net.

The net should also have a continuous (input:output) mapping. Looking at the net as a mapping from an input space q to an output space u, we could actually implement it as a massive lookup table wherein each element represents a tiny hypercube from the input space. As the input varies continuously, the output will vary discontinuously in steps as the input crosses the hypercube boundaries. These steps could be very small but then the size of the table would become too large for economical implementation. If the number of inputs is n and each one is discretized to d levels, the size of the table is d^n. For a modest number of inputs, say 3, discretized into 0.1% steps of their respective domains, the table would require 10^9 entries. We therefore set out to model our continuous mapping by using a continuous neural network, as opposed to a discrete net such as counterpropagation.

Perhaps the greatest difficulty in applying neural networks to real problems is that there is so little mathematical theory to guide us in this area. Most researchers concede to using a trial and error approach in developing neural nets to accomplish specific tasks. Certain network paradigms have received more attention than others from theorists, and backpropagation in particular has associated with it some proofs of convergence and applicability which inspire confidence. Hecht-Nielsen has shown that:

> **Theorem 3.1:** Given any $\epsilon > 0$ and any L_2 function $\mathbf{f} : [0,1]^n \subset R^n \to R^m$, there exists a three-layer backpropagation neural network that can approximate $\mathbf{f}$ to within ϵ mean-squared error accuracy [17].

This result is corroborated by several others such as Stinchcombe and White [42], who have extended this universal approximation property to nets with bounded connection weights.

In theory, we can make the mean-squared modeling error ϵ arbitrarily small by increasing the number of nodes in each hidden layer, without increasing the number of layers. Hecht-Nielsen noted that although three layers are always sufficient, more layers may be desirable for some problems which would otherwise require an intractably large number of nodes. In fact, the number of nodes required is entirely problem dependent, and although 'hard' lower bounds for this number have been proposed [43], we are not aware of any formal procedure for determining an optimal number. Generally, some experimentation is required in designing a network which satisfactorily balances the mutually opposed design criteria of error minimization and network simplicity.

The backpropagation learning algorithm involves updating the internal connection weights by the 'Generalized Delta Rule' described in [44]. When this rule is applied to a feedforward net with semilinear activation functions, the following has been shown to hold:

Theorem 3.2: The generalized delta rule essentially implements gradient descent in sum-squared error for semilinear activation functions [44].

This means that at every learning step, the net's modeling error is reduced (unless we have reached a minimum, in which case the error remains unchanged). Multi-layer nets may contain local minima in addition to the global minimum, so the learning may in theory get "stuck" at a sub-optimal local minimum. However, empirical investigations of local error minima in practical problems indicate that there is little cause for concern [44, 45].

Unfortunately, this theorem does not indicate how quickly the training will converge. The rate of convergence depends on a large number of factors including the initial connection weights, the learning rate coefficients, and the distribution and order of presentation of the training data. Some experimentation is often required to achieve an acceptable rate of convergence.

Although these theorems do not guarantee success with the backpropagation network, they do provide us with some useful guidelines for applying the backpropagation net to a given problem. Theorem 3.1 gives us some insight into the effects of topology on modeling accuracy. Theorem 3.2 guarantees that continued training will never result in increased modeling error provided that the training data consistently represents a single L_2 function.

3.3 Neural Net Inputs and Outputs

A robot's direct dynamics, ie., its simulation can be represented as a system whose inputs are the joint torques, one per joint, and whose outputs the joint positions. It would therefore seem to follow that to model the inverse dynamics, our neural net should simply have the inputs and outputs reversed. In other words, the net inputs would be joint positions, and the net outputs would be joint torques. Because this relation between joint positions and torques is dynamical, any device used to implement it (in this case a neural net) must also be dynamical. The net would need the ability to internally compute the derivatives of its inputs and/or outputs. Although the dynamical net discussed in Section 3.2 has this ability, the backpropagation net does not.

The backpropagation net, once it is fully trained and its weights are fixed, performs a strictly static mapping from its inputs to its outputs. There is a tendency to think of such nets as dynamical because of their ability to learn or self-adapt. The learning process is dynamical, but the dynamics of learning are not directly related to the dynamics (if any) of the relation being learned. In fact, the dynamics of learning are defined only by the backpropagation algorithm, by the node activation functions, and by the learning rate constants, all of which are pre-selected by the designer of the net. The net's dynamics, therefore, are not themselves learned, but instead are only used to learn some unknown static mapping as quickly as possible. How then can we model a robot's inverse dynamics, using only the static mapping of a backpropagation net?

We begin by considering a single-input nonlinear dynamical system, described by:

$$\begin{aligned}
\dot{x}_1 &= f_1(x_1, x_2, \ldots, x_n, u) \\
\dot{x}_2 &= f_2(x_1, x_2, \ldots, x_n, u) \\
&\vdots \\
\dot{x}_n &= f_n(x_1, x_2, \ldots, x_n, u)
\end{aligned} \tag{3.1}$$

where	$x_1 \ldots x_n$	*are the states*
	u	*is the driving torque*
	$f_1 \ldots f_n$	*are nonlinear functions of* $x_1 \ldots x_n$ *and* u
	n	*is the order of the system*

In systems where the states $x_1 \ldots x_n$, and their derivatives are measurable, it is possible to obtain a static inverse mapping of the form

$$u = f_{inv}(x_1, x_2, \ldots, x_n, \dot{x}_i) \tag{3.2}$$

where i can be the index of any one row in Equation 3.1, which contains u in the right-hand side. The mapping f_{inv} can then be approximated by training a backpropagation network of suitable complexity. This net will have at most $n + 1$ inputs, and a single output. If the output y of our original system (3.1) happened to be one of the states $x_1 \ldots x_n$, say x_j, then this static net could actually be thought of as modeling the inverse dynamics relation from y to u, by tapping into the dynamics of the plant itself for the additional n inputs, $x_k, (k = 1, 2 \ldots n, k \neq j)$, and $\dot{x}_i$, as shown in Figure 3.2.

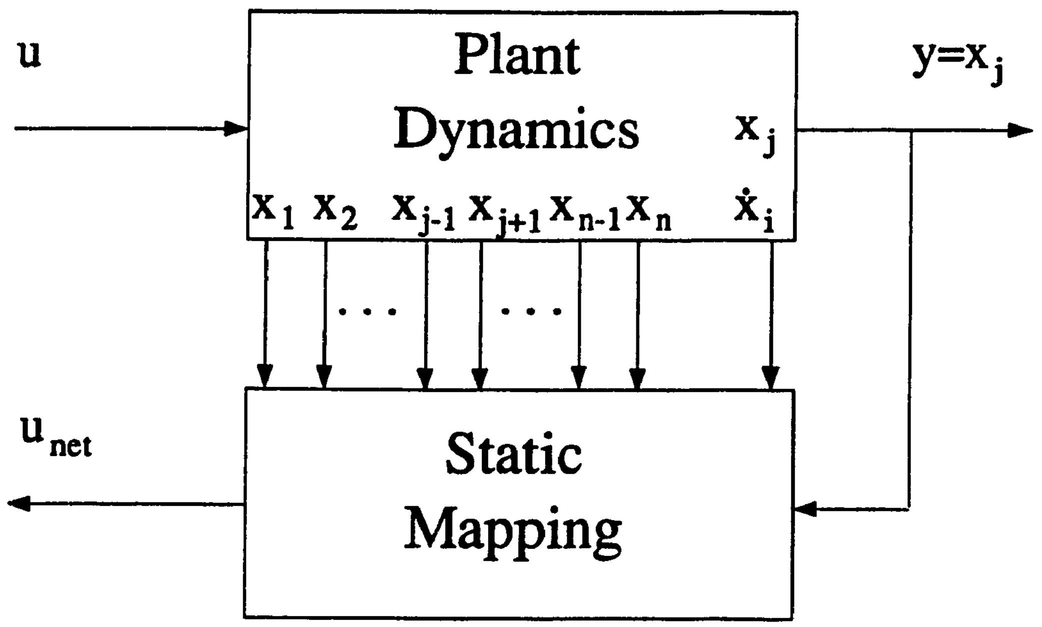


Figure 3.2: Using a static mapping to model the inverse dynamics. (single-input, single-output case)

This approach, although theoretically quite straightforward, has some implementational difficulties that may be insurmountable in all but the simplest of cases. First, the states must all be measurable. This can only be done if the states represent real quantities, and sensors can be obtained or designed to monitor them. Second, although Equation 3.2 indicates that only one of the state derivatives ($\dot{x}_i$) is required, we do not know which one. Actually, if we return to our assumption of no knowledge about the system equations, there is no direct way to proceed with this approach. However, by performing nonlinear transformations on the system, we may be able to map the states $x_1 \ldots x_n$ onto some measurable outputs which have a real-world significance. For example, Spong *et al.* [7] describe the single flexible joint using the state vector

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} q \\ \dot{q} \\ z \\ \dot{z} \end{bmatrix}$$

where : q *is the joint position*

z *is the torque within the joint*

The joint position q and velocity $\dot{q}$ can be readily measured. The joint torque z is more difficult to measure, although sensors to do so have been designed [46]. In addition, it can be shown that our static inverse mapping function would require the measurement of $\dot{z}$ and $\ddot{z}$ (which may be extremely difficult) to yield the relation

$$u = f_{inv}(q, \dot{q}, z, \dot{z}, \ddot{z})$$

Any suitable transformation can be performed on the system equations to obtain a more useful set of state variables. Regardless of the transformation, the resulting inverse static mapping for the n-th order dynamical plant will require $n + 1$ independent input variables, taken from the plant itself.

In this research work, we attempt to develop a general approach to modeling inverse dynamics, yet one which is specifically intended for robotics. Because the joint position is invariably taken as one of the outputs (if not the only output) of the robot, and because it can be measured relatively easily and accurately, we use it as the basic variable in our general approach. Let us consider a real-world single-joint robot of some unknown order n (due to dynamic uncertainties), described by

$$\begin{aligned}
\dot{q}_1 &= q_2 \\
\dot{q}_2 &= q_3 \\
&\vdots \\
\dot{q}_{n-1} &= q_n \\
\dot{q}_n &= f(q_1, q_2, \dots q_n, u)
\end{aligned} \tag{3.3}$$

where : u *is the input*

q_i $i = 1, \dots, n$ *are the states*

$f(\cdot)$ *is a nonlinear (static) function of* $q_1 \dots q_n$ *and* u

The representation given by Equation 3.3 can be obtained from the general system description in Equation 3.1, provided certain conditions are met [3]. From the last row of Equation 3.3, it can be seen that our static inverse mapping will be of the form

$$u = f_{inv}\left(q, \dot{q}, \ddot{q}, \ldots, q^{(n)}\right)$$

where $q \equiv q_1$ (Figure 3.3).

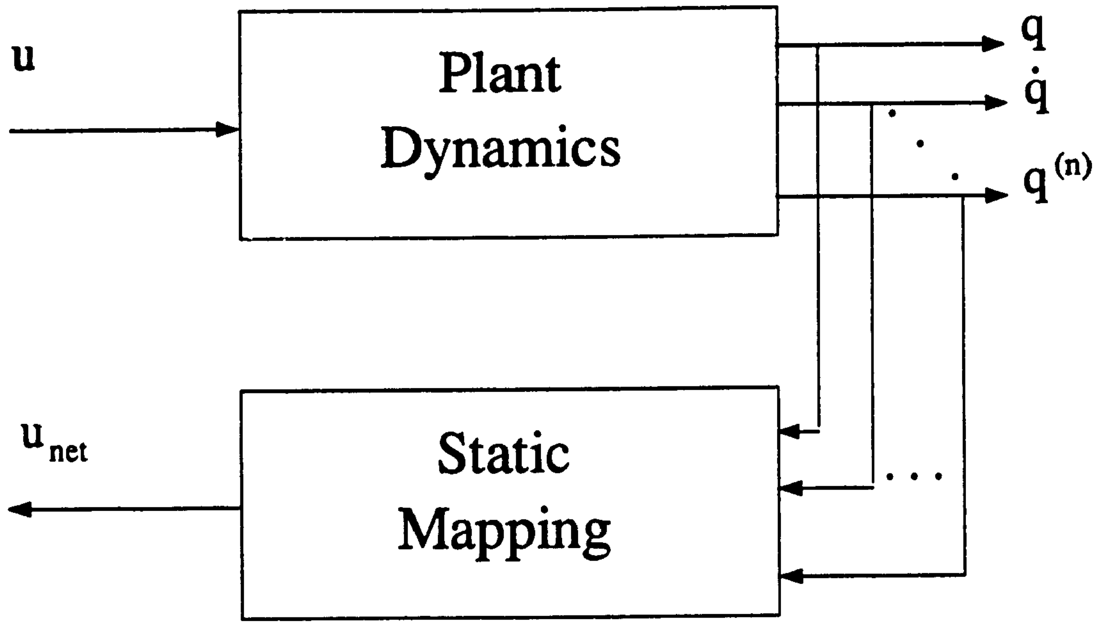


Figure 3.3: Generalization of our approach to modeling inverse dynamics.

Provided that this static mapping represents an L_2 function (a fairly lenient requirement), then a backpropagation net can, in theory, be trained to approximate it. The strength of this approach stems from the fact that we do not require any prior knowledge of $f_{inv}(\cdot)$, nor of the transformation used to obtain the form in Equation 3.3, nor even of the plant's original dynamical equations. We do, however, require a knowledge of the order of the system and the ability to obtain accurate measurements of $q, \dot{q}, \ldots q^{(n)}$.

The ability to measure higher derivatives of q is primarily a technological issue of sensor design. There are significant noise problems to be addressed when taking time derivatives of the signals, and we are not aware of any existing devices for measuring $y^{(3)}$ or $y^{(4)}$. However, for the purposes of the present theoretical investigation, we will assume that the required derivatives (of joint position) can somehow be obtained.

In the case of a multiple-link manipulator in which each joint i is described by an ordinary differential equation of order n_i, the total order of the system will be

$$n = \sum_{i=1}^{N} n_i$$

where N is the number of joints. The design requires at least the first n_i derivatives of the position of each joint i, so the system designer must estimate the value of n_i for each joint. This can either be done intuitively, based on a knowledge of how the robot is built, or experimentally by measuring its response to various types of inputs.

If the neural net is trained with fewer than $n + 1$ inputs, the training procedure will generally not converge. For example, if we assume that the manipulator is rigid and that there are no dynamic uncertainties, then $n = 2$, and the inverse static mapping to be performed by the net is given by

$$u = f_{inv}(q, \dot{q}, \ddot{q})$$

This is consistent with the standard form of the rigid manipulator dynamics given by Equation 1.5. The f_{inv} mapping is fully determined by the three net inputs $q, \dot{q}, \ddot{q}$. In other words, for each input vector $(q, \dot{q}, \ddot{q})$ there is a unique output (u). During supervised training, the net would always be presented with the same desired output vector for any given input vector. Under this condition, learning would converge to some error minimum.

If, on the other hand, the system designer incorrectly estimates that this robot is of first order, and uses only position and velocity as inputs to the net, then the corresponding static net mapping will be given by

$$u = f_{inv}(q, \dot{q})$$

In this case, for a given input vector $(q, \dot{q})$, the net may receive any number of different desired output values (u_d) which depend on the unknown values of $\ddot{q}$. Consequently, the error surface will change at each step of training, and the learning algorithm will be unable to converge.

It should be noted that the system designer can, however, overestimate the order of the robot with impunity. In our example, if a third order system is assumed, then four inputs to the net are required, with the resulting mapping given by

$$u = f_{inv}\left(q, \dot{q}, \ddot{q}, q^{(3)}\right)$$

where $q^{(3)}$ denotes the third derivative of q with respect to t (jerk). Due to the lack of correlation between $q^{(3)}$ and u, the additional input will be treated much like a pure noise input. The backpropagation training algorithm will eventually reduce the weights on all connections to $q^{(3)}$ until this redundant input ceases to affect the net's output. This observation would seem to encourage the designer to use a few more derivatives than deemed necessary, just to be on the safe side. However, it is highly desirable to minimize the number of derivatives used, because of the measurement difficulties discussed previously, and also for the purpose of reducing network complexity and training time. In general, the designer will have to make an accurate estimate of the exact order of the plant. If training does not appear to be converging for a particular net, an additional input derivative may have to be added.

3.4 The "Many-to-One" Mapping Condition

Due to the nonlinearity of the f_{inv} mapping, a "non-uniqueness" condition can cause learning difficulties similar to those caused by underestimating the order of the system. Let us consider the nonlinear first order mapping

$$u^2 = q + 2\dot{q}$$

which can be rewritten in our standard form

$$u = f_{inv}(q, \dot{q}) = \pm\sqrt{q + 2\dot{q}}$$

Here, for every net input, there are two desired outputs which will satisfy the mapping. If, during training, the desired output u_d is sometimes positive and sometimes negative, the backpropagation algorithm will not converge to a solution, but rather will tend towards an "average" output (in this case, zero). To overcome this, we could ensure either $u_d \geq 0$ or $u_d \leq 0$ at all times during training. However, the choice of any such restrictions would generally have to be based on the plant dynamics which we assume are unknown.[2]

In general, backpropagation is capable of learning input/output mappings that are one-to-one or many-to-one, but not one-to-many. Intuitively, we would not expect to encounter a one-to-many mapping for a non-redundant real robot, because such a mapping would imply that different torque trajectories could be used to yield any single end-effector trajectory. Although we have never encountered a robot that exhibits such behavior, the theoretical possibility nevertheless exists, and this risk must be acknowledged. In this work, however, we assume a one-to-one or many-to-one mapping. Should learning fail to converge for a given robot after the previously-discussed precautions have been taken, then this assumption should be re-examined for the robot in question.

3.5 Network Topology

In our simulation, we have compared several different network configurations. One configuration which provided a good compromise between modeling accuracy and total learning time is shown in Figure 3.4. The input buffer layer contains strictly linear activation functions and has the fixed weights shown in Table 3.1. This layer does not

[2] There is an additional constraint imposed by this plant's dynamics that $(q + 2\dot{q}) \geq 0$. This is ensured when q and $\dot{q}$ are taken directly from the plant, however if these net inputs are to be generated by some other means, then the preceding condition would have to be checked explicitly.

perform any learning. It is used only to scale the net input signals so that Layer 1 nodes are not initially driven to the extreme ends of their activation functions.[3] Although the scaling is not necessary, it does significantly reduce the total learning time for the net. The weights have been estimated based on the observed operating range of the robot's outputs (net inputs).

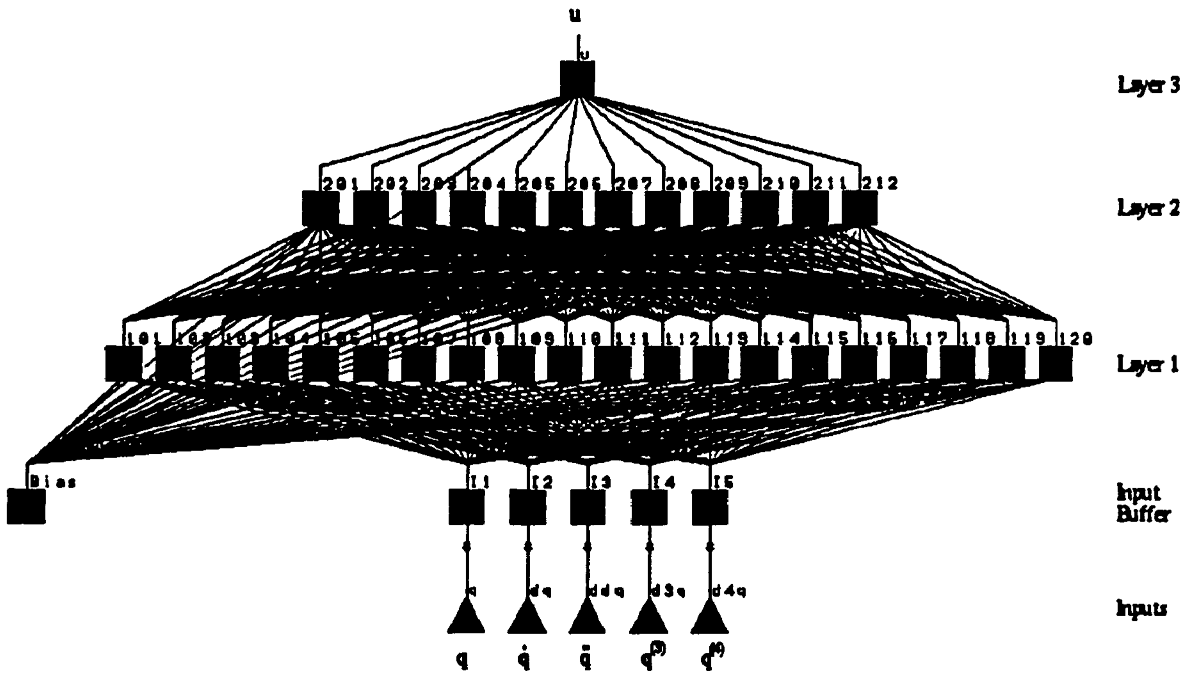


Figure 3.4: Topology of neural net used in simulations.

[3] Since the learning rate is proportional to the derivative of the activation function, learning with sigmoid and hyperbolic tangent activation functions is fastest when the sum of a node's inputs is close to zero.

input	weight
q	0.33
dq	0.04
ddq	0.01
d3q	0.0033
d4q	0.0002

Table 3.1: Fixed weights of nodes in Input Buffer Layer.

Each node in Layers 1–3 is driven by all the nodes of the layer immediately below. An additional bias input has been added to each node to allow for the possibility of a fixed offset.[4] The input function of any node j is the weighted summation

$$A_j = \sum_{k \in K} w_{jk} I_{jk} + w_{jb} B$$

$$\begin{array}{lll} where: & K & is\ the\ set\ of\ all\ nodes\ in\ the\ layer\ below\ node\ j \\ & B & is\ the\ output\ of\ the\ bias\ node\ (B = 1) \\ & b & is\ the\ index\ of\ the\ bias\ node \end{array}$$

The output O_j of node j is related to its activation A_j by the output function

$$O_j = tanh(A_j)$$

Although the standard backpropagation algorithm uses the sigmoid output function whose outputs are in the range (0,1), we use the hyperbolic tangent to obtain a bipolar output in the range (-1,1). This is a valid substitution because both of these functions are semilinear, a necessary condition for successful application of the backpropagation algorithm as defined by Rumelhart *et al.* [26].

[4] For example, if all net inputs are zero and the desired net output is non-zero a bias signal is required for trainability.

Layer 3, the output layer, consists of a single node which maps the outputs of Layer 2 into the overall net output u. In the multiple-joint case, the net would have one input node for each joint in the manipulator.

Because the net's output corresponds to the manipulator's driving torque, the range of this output will be the same as the operating range of the manipulator (after the net has been trained). However, the hyperbolic tangent output function limits the net output to the range (-1,+1) which generally falls far short of any real manipulator's operating range. We overcome this problem by scaling (multiplying) the net's output by the actuator torque limits. Doing so gives the net the ability to generate the required output torques, while guaranteeing that the net cannot under any circumstances damage the robot it is driving by exceeding its torque limits.

A potential shortcoming of this approach becomes manifest when the robot is not trained (but operates) over its entire permissible torque range, as would often be the case. The ability of the net to model nonlinear mappings is given by the nonlinear node output functions, in our case, the hyperbolic tangent function. The derivative of $u = U_{max} tanh\,(x)$ varies from $\dot{u} = U_{max}$ when $u = 0$ to $\dot{u} = 0$ as $u \longrightarrow \pm U_{max}$, where U_{max} is the given robot's actuator torque limit. This significant variation in $\dot{u}$ provides a measure of the degree of nonlinearity of the hyperbolic function. By contrast, if the robot were used only within one fifth of its maximum actuator torque range, $\dot{u}$ would vary only from $\dot{u} = U_{max}$ when $u = 0$ to $\dot{u} = 0.96 U_{max}$ when $u = U_{max}/5$. In other words, $\dot{u}$ would remain nearly constant over the operating range, thus limiting the Layer 3 output mapping to an almost linear relationship. Recalling that we need three <u>nonlinear</u> layers in the net, Theorem 3.1 provides strong motivation to add a fourth layer to the net. Once this is done, the single-node output layer can implement a strictly linear mapping function, however we retain the hyperbolic tangent function with appropriate output scaling to protect against exceeding actuator limits. The effects of varying operating range and number of layers are illustrated empirically by the simulations in Chapter 5.

3.6 The Generalized Learning Model

As we have discussed, the fact that the backpropagation net uses supervised learning implies the need for a training signal. In the past, most researchers [14, 19, 20] using neural nets to model robot inverse dynamics have adopted an approach referred to by Psaltis *et al.* [21] as specialized learning. This approach, illustrated in Figure 3.5, consists of driving the unknown system with a neural net while using the error measured at the system's output to update the internal weights of the net. The success of this method has been demonstrated for rigid systems. However, we feel that it is unsuitable for flexible manipulators. Because highly flexible joints tend to be only marginally stable, their linearized models have a complex conjugate pair of poles close to the imaginary axis of the s-plane [7]. During the early stages of learning, the weight changes taking place within the neural net tend to reinforce the oscillatory tendency of the plant.

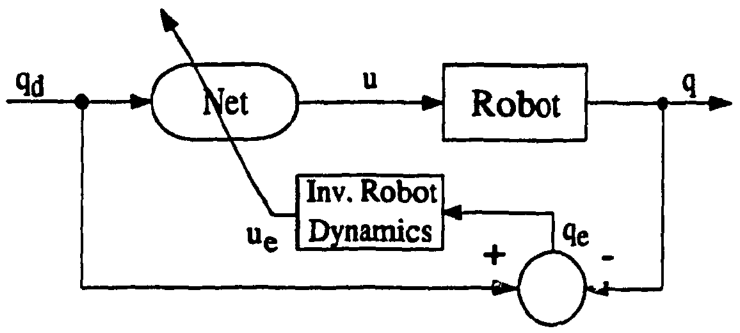


Figure 3.5: Specialized learning configuration.

Another difficulty that arises with specialized learning is the question of how to determine the training signal u_e that corresponds to the measured system output error q_e. This correspondence, given by the Jacobian of the plant, is unavailable to us under our assumption of unknown plant dynamics. Kawato *et al.* [20] have estimated it using constant values. Chen and Pao [19] proposed an adaptive method called the inverse transfer learning scheme. Both of these approaches however are only approximations, so some error is inevitably introduced into the learning procedure.

In order to avoid these problems, we have adopted the generalized learning model [21] as shown in Figure 3.6. In this scheme, the learning procedure cannot cause unwanted plant oscillations, because the plant is no longer driven by the net. Furthermore, the torque error is given directly and accurately. The main disadvantage of generalized learning is that it is an off-line procedure. The net is first trained at the output of the plant as shown in Figure 3.6. Once training has been completed the weights are fixed and the net is placed in front of the plant to drive it . It should be noted that the two types of learning could be combined by first performing generalized learning, to establish an accurate inverse dynamics model and then running the system under specialized learning with a slower learning rate so as to compensate for variations in the plant dynamics.

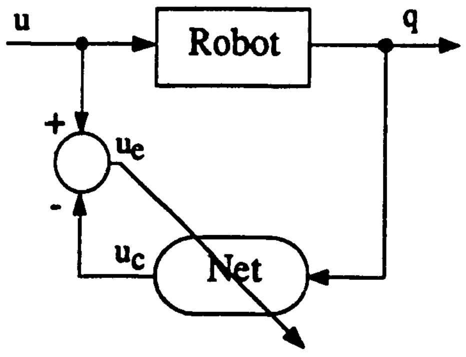


Figure 3.6: Generalized learning configuration.

3.7 Lowpass Noise Training

The generalized learning model requires that we provide torque input trajectories during learning. Our goal is to train the net so that it performs well for any arbitrary joint space trajectory (subject to actuator constraints) that we may later present to it. The most general type of training signal would be white noise. However, since the expected system input is limited to low frequencies, only lowpass noise is used. To simulate this signal, we employ a summation of randomly chosen sinusoids given by:

$$u(t) = \sum_{i=1}^{N} A_i \sin(\omega_i t) \tag{3.4}$$

where $N = 5$. A_i and ω_i are real constants generated pseudo-randomly within the ranges

$$-A_{max} \le A_i \le A_{max}$$
$$0 \le \omega_i \le \omega_{max}$$

and updated at 10 second intervals during learning. A_{max} is chosen so that $u(t)$ remains within actuator constraints. Clearly, our bounding method is vastly oversimplified, mainly because in simulation there is no danger of damaging any hardware. It is understood that when a real robot is used, a more reliable method for remaining within actuator limits will be required. Several simulations with different values of ω_{max} were carried out, resulting in widely varying degrees of learning ability as discussed in Chapter 5.

Using a simulated noise signal generates a uniformly populated input space during learning. As a result, during subsequent network recall, any input vectors presented will be relatively close to some of the vectors used during training; the distances over which the net has to interpolate will remain small [21]. The noise signal also results in an unordered presentation of the inputs during learning. This is a highly desirable condition for convergence to a global minimum when using, as we do, the generalized delta training rule.

It is very important for the training signal to exercise the entire operating range of the robot, in terms of joint positions, velocities, and all applicable higher derivatives. If the intended operating range is bounded only by the actuator limits, then the training signal must uniformly exercise the full range of torques between those limits. On the other hand, if we know in advance that the robot will only be used for a set of operations that exercise a portion of its maximum operating range, then it is more advantageous to train only within that smaller range. Doing so will result in a denser training set distribution over a smaller (generally simpler) subspace. Given the same net and the same training time, significant reduction in modeling error can be achieved.

3.8 Learning Rates

Recalling the generalized delta learning rule (Equations 2.1 – 2.3),

$$\Delta w_{jk} = c_1 * \delta_j * I_{jk} \tag{3.5}$$

$$\delta_j = (D_j - O_j) * f_j'(A_j)$$

$$\delta_j = f_j'(A_j) * \sum_{q \in Q} (\delta_q * w_{qj})$$

we note that the learning rate c_1 is used to scale the amount by which the node input weights change at each learning step. The exact value of c_1 is chosen by the net designer and is generally less than 1.0.

A very low learning rate yields relatively slow convergence but a smooth trajectory of descent along the error surface[5]. Higher learning rates will generally yield faster initial

[5] Assuming random order of presentation of the input data.

convergence into some neighborhood around the error minimum, with no further average error reduction after that. The weights tend to fluctuate a great deal with higher learning rates, resulting in a jagged trajectory along the error surface. This tendency can cause overall system instability if the net is being trained on-line and its output is used to drive the plant. Under such conditions, low values of c_1 should be used to minimize the effects of the net learning dynamics on the overall system dynamics. In generalized training, this is not a concern. In fact, large weight fluctuations are desirable because they help the network escape local minima and flat (almost zero gradient) regions on the error surface. Conversely, there exists the danger of jumping from the global minimum's region of convergence into a local minimum's region of convergence; however, the probability of going in the opposite direction is greater[6].

It can be seen from the generalized delta rule equations that even if the net attains its exact global error minimum, it will not remain at that minimum unless that point corresponds to exactly zero error. In general, a network of finite size cannot model an arbitrary mapping exactly but can only approximate it to within some non-zero error ϵ. This error will cause the network to move away from its error minimum at the next learning iteration, and then to move back towards the minimum at the subsequent iteration. Because the step taken at each iteration is proportional to the learning rate, a smaller value of c_1 at this point will tend to maintain the net within a closer proximity of its error minimum.

We employ the advantages of both fast and slow learning rates by the process of simulated annealing [37, 47]. At the commencement of learning, the network state is relatively far from the minimum error state. We want the network to progress quickly toward the global error minimum, while escaping from any local minima and flat regions

[6] The amount by which the weights change is proportional to the error, and local minima correspond to a higher error than the global minimum, so jumps from within the local minimum's region of convergence will on average be larger than jumps from within the global minimum's region of convergence.

that may be encountered along the way. The learning rate is therefore initially set high. As learning progresses and the net approaches its minimum error state, we gradually reduce the learning rate to smooth the path down the error surface and reduce the amount of overshoot of the minimum. The learning rate is, therefore, set as a function of the accumulated learning time according to some predetermined "annealing schedule". A typical schedule used in many of our simulations is shown in Table 3.2.

Learning Step:	1 to 5000	5001 to 20000	20001 to 50000	50000 +
Layer 1:	1.00	0.50	0.25	0.13
Layer 2:	0.50	0.25	0.13	0.06
Layer 3:	0.25	0.13	0.06	0.03

Table 3.2: Learning rate (c_1) schedule for a 3-layer net.

Although it is possible to have an independent learning rate for each node, or to have the same learning rate for all nodes in the net, we have found it most advantageous to have an independent rate for each layer. The fastest rate is in the lowest layer, with increasingly slower rates for successively higher layers.

3.9 Initialization and Timing

Once a network's topology and learning rates have been established, all of its interconnection weights must be initialized to some non-zero values prior to the commencement of learning. As can be seen from Equation 3.5, the amount of weight change Δw_{jk} at each learning step is directly proportional to the node's input I_{jk} along that connection. If a given connection weight w_{jk} is initially zero, there will be no input along that connection ($I_{jk} = 0$), so the corresponding amount of weight change (Δw_{jk}) will be zero, and w_{jk} will remain at zero forever. All interconnection weights must therefore be initialized to non-zero values.

If, however, all the weights in the net were initialized to the same non-zero value, it can be seen that even as learning progresses, all the nodes in any given hidden layer would always have equal outputs[7]. Such a network could be collapsed to a net with the same number of layers but with only one node in each hidden layer; any additional hidden layers would be redundant. However, we need a net for which the addition of hidden nodes will tend to enhance its modeling capability. The initial weights must, therefore, be non-zero and unequal. The weights in our simulations are initialized independently by a pseudo-random number generator producing values in the range $[-0.1, 0.1]$.

During learning, the net is simulated as a discrete, synchronous sub-system with a fixed sampling rate. The inputs and the desired outputs are held constant for the duration of each learning step, while the net recomputes its internal connection weights which become effective at the beginning of the next time step. The simulated system can be thought of as continuous-time net with synchronized zero-order "sample and hold" devices at its inputs and its outputs. As a result, the dynamics of signal propagation through the net do not affect its output signal, and transient states are avoided.

No attempt has been made to run these simulations in real time, so we placed no restrictions on the size of the time step. For real-time simulation or hardware implementation, there would certainly be a lower bound on the step size. We use a step size of 0.02 sec., which for real-time feedforward operation of the net in Figure 3.4 would entail approximately 0.5 MFLOPS, a rate easily attainable by many off-the-shelf floating point co-processor chips.

[7] Based on the assumption that all nodes in a given layer have the same learning rate and the same output function.

3.10 Summary

A method has been presented for using a backpropagation net to model the inverse dynamics of the flexible-joint manipulator. We have shown that even a static net is capable of approximating a robot's driving torques directly from the joint positions and their derivatives, provided that a sufficient number of those derivatives are obtainable. A procedure for training such a net has been detailed..

This neural network based approach has the distinct advantage of allowing us to develop controllers for robots whose dynamics are unknown. However, the control system designer must know the order n_i of the differential equation describing the dynamics corresponding to each joint i of the manipulator, and be able to accurately measure (or compute) at least n_i derivatives of the position of this joint.

In the next chapter, we show how such a net can then be used as part of a nonlinear feedb [illegible] ontrol system. The following chapter shows an application of our approach to a [illegible] -of-freedom flexible-joint manipulator. Both the network training and on-line contr [illegible] ses are evaluated by computer simulation.

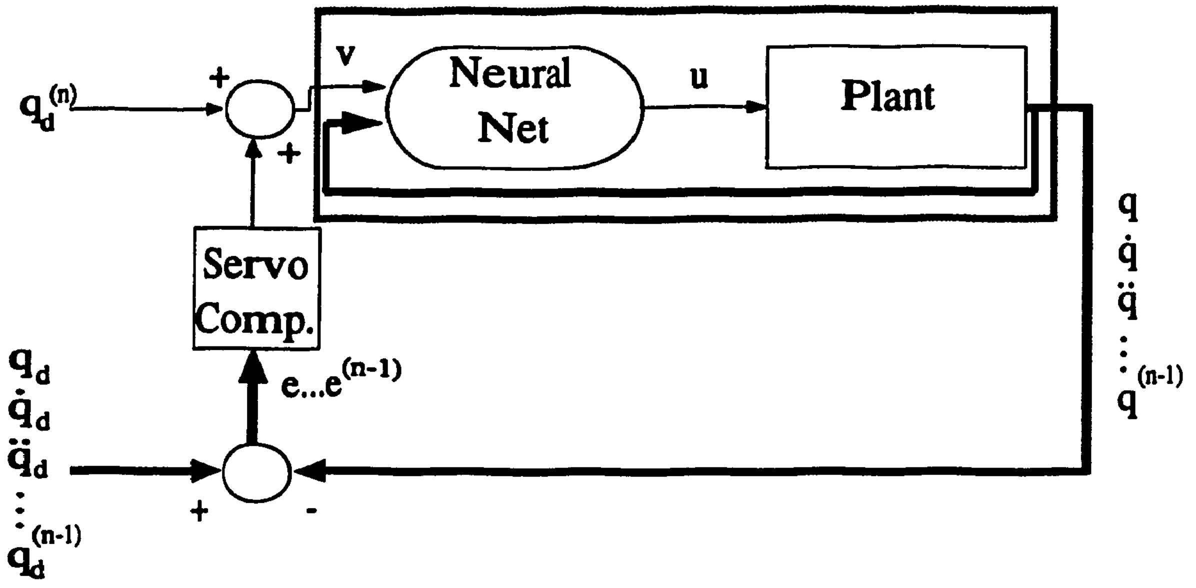
4
THE PROPOSED CONTROL SYSTEM

4.1 System Structure

As discussed in Chapter 1, we seek to satisfy two major design objectives. First, the controller design must be attainable without the availability of the robot's dynamical equations. Second, the overall system response must conform precisely to some independently-provided performance specification[8].

These potentially conflicting goals can simultaneously be satisfied by partitioning the controller as shown in Figure 4.1. The inner loop has the sole responsibility of linearizing the plant. We use a neural net that has previously been trained off-line as described in Chapter 3. For a single-joint manipulator with a dynamical model of order n, this net will have $n+1$ inputs and a single output. The first n inputs $(q, \dot{q}, \ddot{q}, \ldots, q^{(n-1)})$ remain connected to the plant, just as they were during training. The highest order net input, which was connect to the plant output $q^{(n)}$ during training, is now connected to the outer loop feedback signal v, whose units must therefore be the same as those of $q^{(n)}$. The inner loop is closed by connecting the net's output u to the robot's input (driving torque).

[8] The specification must, of course, be within the physical limitations of the robot actuators and the controller hardware. This problem was examined in Section 1.3.



$q_d^{(n)}$
+
+
v
Neural Net
u
Plant
Servo Comp.
e...e$^{(n-1)}$
q_d $\dot{q}_d$ $\ddot{q}_d$ $\vdots$ $q_d^{(n-1)}$
+
-
q $\dot{q}$ $\ddot{q}$ $\vdots$ $q^{(n-1)}$

Figure 4.1: The control structure for an arbitrary n-th order, single-joint robot.

As a consequence of the "many-to-one" mapping condition described in Section 3.4, any given net input vector $\left(q, \dot{q}, \ddot{q}, \ldots, q^{(n-1)}, v\right)$ will produce a unique net output u. Let us now make the assumption that for any robot state $\left(q, \dot{q}, \ddot{q}, \ldots, q^{(n-1)}\right)$, the driving torque u uniquely determines the rate of change of state $q^{(n)}$. This is equivalent to assuming that the robot is controllable. Under this assumption, there exists a one-to-one mapping between u and $q^{(n)}$ for any given robot state $\left(q, \dot{q}, \ddot{q}, \ldots, q^{(n-1)}\right)$. Since the net was trained with its $(n+1)$th input connected to $q^{(n)}$, the trained net will, therefore, produce an approximation of that driving torque u which would result in the plant output $q^{(n)}$ being equal to the net's $(n+1)$th input. By connecting the net's $(n+1)$th input to the feedback control signal v, and leaving all the other inputs connected to the robot, the net output u will approximate the driving torque required to produce the plant output $q^{(n)} = v$. The dashed box enclosing the inner feedback loop in Figure 4.1 can therefore be treated as a sub-system described by

$$q^{(n)} \approx v, \tag{4.1}$$

where the quality of the approximation depends exclusively on the modeling error of the net[9]. In the theoretical limiting case of a perfectly-trained net, this subsystem is not only linear, but represents nothing more than a simple n-th order integrator. The trained net and the inner loop together form what we will call the model-based portion of the controller, so named because it models the inverse dynamics of the robot.

The second portion of the controller is the outer loop, commonly referred to as the servo portion. The servo design does not concern itself with the robot dynamics; instead it treats the model-based portion as a black box which is assumed to perform n-th order integration. Only the order of the robot is needed for the servo design. Any linear feedback strategy can be applied to the outer loop. Our approach is a straightforward extension of constant-gain PD control. Defining the servo error as $e = q_d - q$, and assuming that Equation 4.1 is a good approximation, the equation of motion for the overall n-th order system becomes

$$e^{(n)} + k_{n-1}e^{(n-1)} + \ldots + k_1\dot{e} + k_0 e = 0 \tag{4.2}$$

where $k_0, k_1, \ldots, k_{n-1}$ are the feedback gain constants which can be set to provide any desired closed-loop system poles without any specific knowledge of the plant dynamics. The resulting control law is given by

$$v = q_d^{(n)} + k_{n-1}\left(q_d^{(n-1)} - q^{(n-1)}\right) + \ldots + k_1\left(\dot{q}_d - \dot{q}\right) + k_0\left(q_d - q\right)$$

This partitioned control strategy satisfies both of our major design objectives. By employing a neural network, previously trained off-line by the procedure described in Chapter 3, the model-based portion compensates for the robot dynamics without requiring any knowledge of the robot's specific dynamical equations. The servo-based portion can then be designed to yield some pre-specified overall system response, also independently of the specific robot dynamics.

[9] Negligible inner-loop time delays are assumed.

4.2 Comparison with Feedback Linearization

A comparison of Figures 4.1 and 1.1 reveals some similarities between our controller and the feedback linearization approach for a single flexible-joint manipulator. Both controllers use a partitioned strategy, wherein the inner loop linearizes the plant, and the outer loop provides generalized linear system control. The outer loop (servo portion) is the same for both systems, and both assume the ability to transform the plant outputs into the form:

$$\begin{aligned}
\dot{q}_1 &= q_2 \\
\dot{q}_2 &= q_3 \\
&\vdots \\
\dot{q}_{n-1} &= q_n \\
\dot{q}_n &= g\left(q_1, q_2, \ldots, q_n, u\right)
\end{aligned}$$

Our controller also shares the feedback linearization controller's ability to create a "new" subsystem which is both linear and decoupled [1]. Here, the similarities end.

The model-based portion of conventional feedback linearization systems requires prior knowledge of the exact inverse dynamics of the robot. This knowledge includes not just the order of the robot and the structure of its dynamics equations, but also the actual values of all parameters. By contrast, our system circumvents this dependency by training a neural net to perform that same mapping.

A second marked difference lies in the way that the servo feedback signal v is passed to the robot input u. In feedback linearization,

$$u = f_a v + f_b \tag{4.3}$$

where f_a and f_b are some nonlinear functions of the robot's outputs. For any given plant output vector, f_a and f_b are constant, so u is directly proportional to v. The servo feedback signal is said to enter linearly into the plant. Due to this restrictive condition, Equation 4.3 may fail to linearize certain plants. For example, Cesareo and Marino [5] have shown its inability to linearize a planar 2–DOF flexible-joint manipulator.

In our system, on the other hand, the relationship between u and v is given by

$$u = f_{inv}\left(q, \dot{q}, \ldots, q^{(n-1)}, v\right) \tag{4.4}$$

where f_{inv} is a nonlinear function of both the robot's output $\left(q, \dot{q}, \ldots, q^{(n-1)}\right)$ and of the servo feedback signal v. Consequently, v is not restricted to entering linearly, so Cesareo and Marino's proofs of nonlinearizability do not apply. Although there exists no general proof of linearizability for our neural net based controller, comparison of Equations 4.3 and 4.4 reveals that our controller's modeling capabilities represent a superset of those of conventional feedback linearized controllers.

4.3 Selecting the Outer Loop Gains

When a sufficiently well trained network is obtained for the model-based portion of the controller, Equation 4.2 becomes a good approximation of the overall system response. Under these conditions, it is possible to fully specify some desired system response by setting the servo gain constants appropriately.

We have chosen to specify that all the system poles be placed at a single location on the real axis of the s-plane. For example, in our simulations of the single-link flexible-joint manipulator, the closed-loop response is given by

$$e^{(4)} + k_3 e^{(3)} + k_2 \ddot{e} + k_1 \dot{e} + k_0 e = 0$$

To place the four poles of this system at a point s_p on the real axis, we must use

$$\begin{aligned} k_0 &= s_p^4 \\ k_1 &= -4s_p^3 \\ k_2 &= 6s_p^2 \\ k_3 &= -6s_p \end{aligned}$$

In our simulations, except where otherwise noted, we have chosen $s_p = -10$, resulting in the servo gains $k_0 = 10000$, $k_1 = 4000$, $k_2 = 600$ and $k_3 = 40$.

Although the choice of s_p is up to the designer, it will be limited by both the modeling accuracy of the net and by loop delays in a real system. As s_p is moved farther to the left in the s-plane, the system response becomes faster, but the resulting dramatic increase in feedback gains makes the system much more sensitive to modeling inaccuracies. We will show simulations in which the net was not accurate enough for a given pole placement, and the system became unstable.

We have used a quadruple real pole throughout all the simulations, but this is by no means an inherent restriction of the control system. The designer is free to choose any poles that he/she feels are best suited to a particular application.

4.4 On-Line Training

Although the neural net is not installed in the closed-loop system until it has been satisfactorily trained off-line, it may sometimes be desirable to continue net training on-line. Such training would allow system adaptation to environmental changes and robot dynamics variations. It would also enable adaptation to frequently repeated trajectories. For example, a neural net trained with the white noise generalized training procedure may later be used on-line for only one task which is being repeated continually. On-line training would fine-tune this net to better model specifically those plant dynamics which are most excited by that particular task.

As is always the case with supervised nets, an error signal is required for such on-line training. To obtain this signal, a second net is added to the closed-loop system, as shown in Figure 4.2. The two nets are identical at all times. Recalling that the error signal must be in the same units as the net output, we can obtain this torque error u_e by taking the difference between the actual robot input u and the output of the second neural net u_c. The torque error signal is then used to train Net 2 in exactly the same way that the off-line training was done. In fact, the subsystem formed by Net 2 and the robot in Figure 4.2 is identical to the generalized training configuration presented in Section 3.6. The notable difference is that the robot's input u is no longer a random torque signal, but rather a task-specific feedback signal which depends both on the robot dynamics and on the accuracy with which Net 1 models those dynamics.

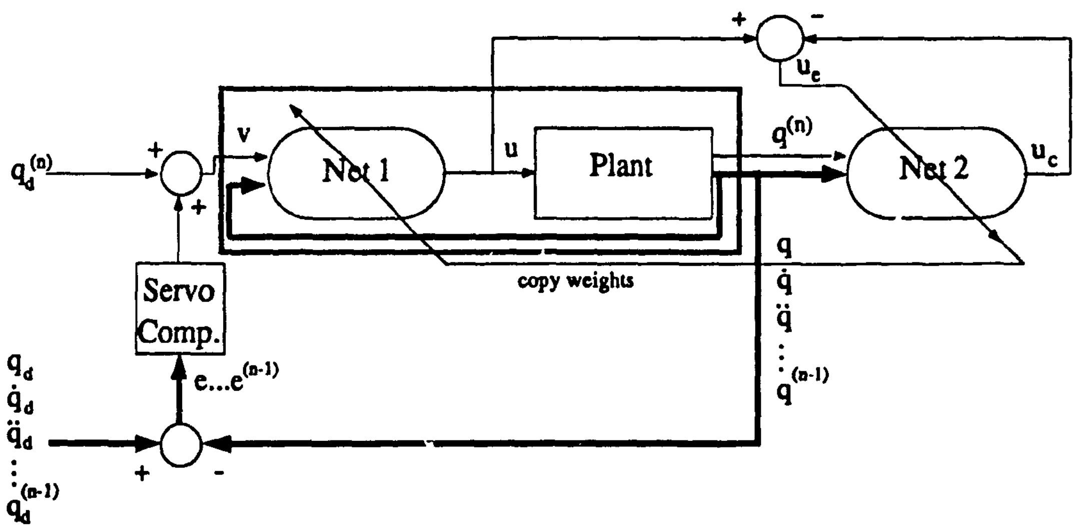


Figure 4.2: A closed-loop controller incorporating on-line training. The two nets are identical.

As the training of Net 2 proceeds and its connection weights are updated, those weights are immediately copied into Net 1 so that the two nets remain identical at all times. In a sense, Net 1 can also be thought of as learning, but not exactly as defined by the backpropagation algorithm. Feeding the error signal u_e directly to Net 1 would not result in the same weight changes, and the two nets would not remain identical. To understand why, let us review the generalized delta rule give by Equations 2.1 – 2.3. The weight change in the output layer would be given by:

$$\Delta w_{jk} = c_1 \times u_e \times f_j'(A_j) \times I_{jk}$$

where both A_j and I_{jk} are functions of the net input after it has been propagated forward through the net. Whenever u_e is non-zero, it follows that the outputs of the two nets are necessarily not equal ($u \neq u_c$). Since the nets are identical, their inputs must be different. Consequently, A_j and I_{jk} will differ from one net to the other, and the weight changes in the output layers of the two nets will not be equal. This line of reasoning can easily be extended to show that the weight changes in any node will generally not be the same for the two nets. Therefore, in order to maintain identical nets, we perform backpropagation training in Net 2, and copy its weights to Net 1.

Net 2 has been chosen as the training net because of the way u_e is obtained from the system. The training error signal must be the difference between the net's actual output and its desired output. Since all of the inputs of Net 2 are taken directly from the robot outputs and the net is supposed to model the inverse mapping of the robot, it is clear that its desired output is simply the robot's input u. By contrast, there exists no suitable signal within the system to provide the desired output for Net 1. If there were such a signal, we could eliminate both nets and use that signal to drive the robot directly.

Having established a configuration for on-line training, it may be tempting to use this configuration from the outset, and to completely eliminate off-line training. This cannot be done in general. Placing an untrained net into the on-line feedback loop, particularly for flexible-joint manipulators with their inherently oscillatory nature, is likely to cause closed-loop instability. The off-line training is needed initially to reduce the modeling error of the net sufficiently for it to be stable when it is later placed in the closed loop system. The higher the servo feedback gains, the more accurate the net must be before it can be placed into the feedback loop.

Another potential danger of on-line training is that the net's internal learning dynamics will reinforce the oscillatory tendency of the robot dynamics, and thereby threaten system stability even when the net models the robot quite accurately. This risk can be reduced by slowing the rate of weight changes in the net, thus introducing a time scale separation between the learning dynamics and the robot dynamics. The net learning rate constants can be used to achieve this slow-down. On-line learning rates should usually be lower than those used for off-line training. Their exact values will depend on both the nature of the robot and on the servo gains.

4.5 The Simulation Tools Used

All neural network and control system simulations were run on a Sun 3/60 workstation. The neural net training was done using a commercial package called NeuralWorks Professional II [48]. The network block diagrams and training error plots were also generated by this package. All other plots were obtained using a robot simulation software package developed in a joint project at Concordia and McGill Universities. The robot and control system simulation software was written in the C language. A 4th/5th order variable time-step Runge-Kutta integration routine [49] was called to solve the set of differential equations used to simulate the robot's direct dynamics.

The behavior of the manipulator was simulated mathematically, using the model described in Chapter 5. The equations describing this model, however, are strictly internal to the software module that performs the robot simulation, and are in no way available to the neural net or used in the system design. The manipulator is treated as a black box about which the only obtainable information is its input/output response. Replacing the simulated manipulator with a physical one, therefore, would have no effect on the design approach taken in this work.

The neural net was also simulated, but using the NeuralWorks package rather than software developed in-house. This package has proven to be extremely helpful in some ways and restrictive in others. It provides graphical tools for laying out and revising the network topology in seconds. Parameters can be edited and one can even change the network paradigm with a few simple mouse operations. Both the net and certain performance graphs can be displayed on the screen or sent to a laser printer. The simulation package provides an ideal environment for the rapid development and testing of new ideas.

Once these ideas have been developed, and a satisfactory net has been obtained, the commercial package may no longer be the most suitable tool. Its versatility and features introduce computational overhead which can be avoided by writing dedicated software for the chosen net. Furthermore, integrating the NeuralWorks package with other software for the purpose of simulating an entire control system of which the net is but one component, can be difficult and in certain cases impossible. For example, the on-line training configuration employing two nets cannot be reasonably simulated using NeuralWorks.

To take advantage of the simulation package's features while avoiding its pitfalls, we approached off-line training quite differently from the closed-loop control system simulations. The NeuralWorks package was used only for the training phase. This package can be set up to take advantage of the UNIX's multitasking capabilities by requesting its input data from another running process, rather than just reading it from an input file. The concurrent process referred to by NeuralWorks as User I/O is written in C and compiled as a separate program, whose sole function is to wait for and respond to NeuralWorks' request for data.

The two processes communicate using UNIX IPC (inter-process communication) shared memory and semaphores, as shown in Figure 4.3. NeuralWorks first spawns User I/O and sets up the communication channel. At each training step (0.02 sec) NeuralWorks requests two sets of data: net inputs and desired net outputs. To provide these data, User I/O must simulate the generalized training configuration of Figure 3.6. When net input data is requested, User I/O computes a new pseudo-random manipulator input torque, advances the manipulator simulation by one learning step by integrating the robot's differential equations, and places the robot's new outputs (q, $\dot{q}$, $\ddot{q}, \ldots$, $q^{(n)}$) into the shared memory. When desired net output data is requested, User I/O places the manipulator's current input torque (u) into the shared memory. By employing User I/O, the training data can be produced "on-the-fly", thus obviating the need to first generate huge data files. This is particularly important because we are using random training signals, rather than repeatedly presenting one training set as is frequently done for backpropagation nets.

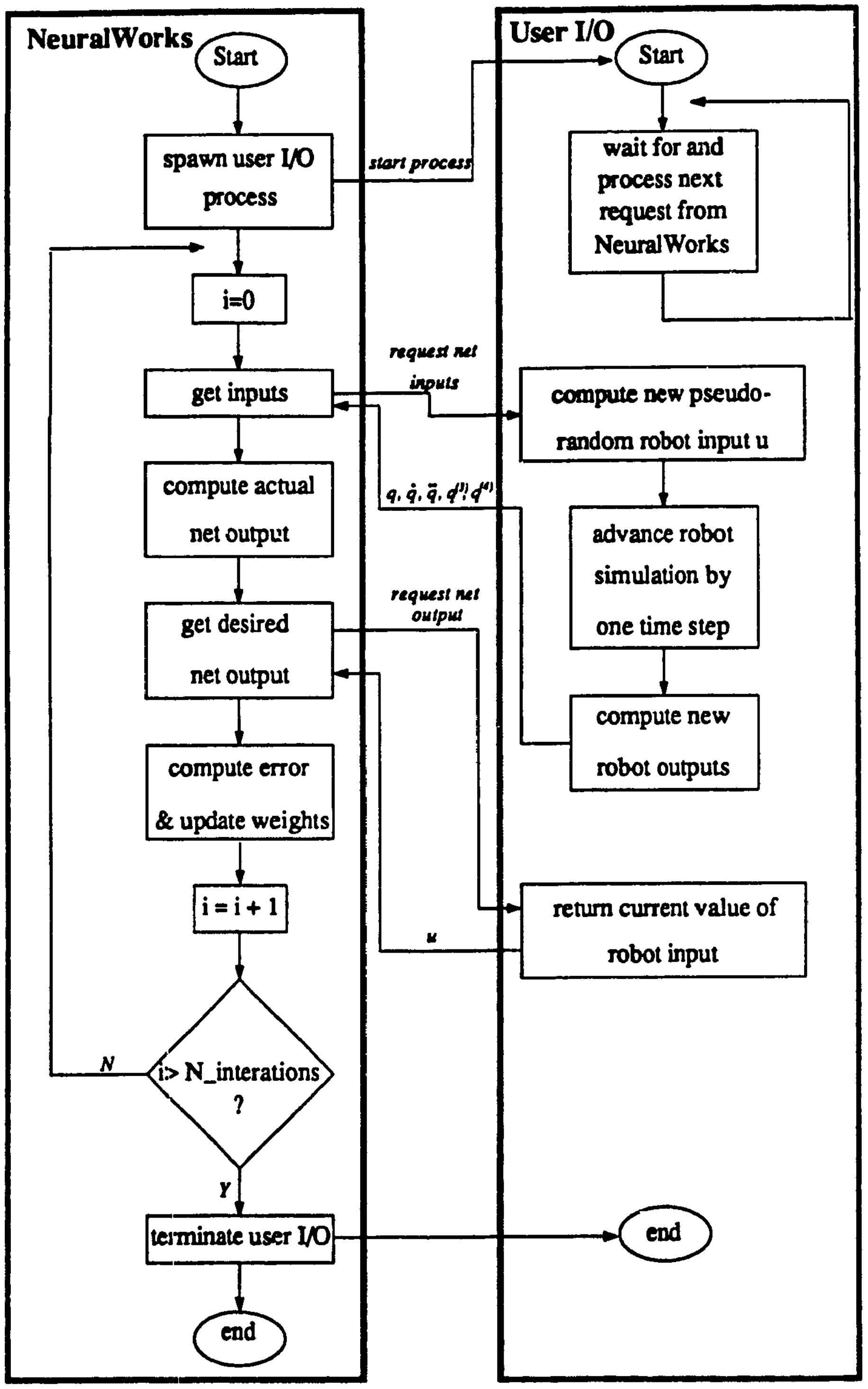

NeuralWorks
Start
spawn user I/O process
i=0
get inputs
compute actual net output
get desired net output
compute error & update weights
i = i + 1
i > N_interations ?
N
Y
terminate user I/O
end
start process
request net inputs
$q, \dot{q}, \ddot{q}, q^{(3)}, q^{(4)}$
request net output
u
User I/O
Start
wait for and process next request from NeuralWorks
compute new pseudo-random robot input u
advance robot simulation by one time step
compute new robot outputs
return current value of robot input
end

Figure 4.3: Interprocess communication between NeuralWorks and User I/O during off-line training.

Simulation of the closed-loop control system is done quite differently. Because we use a variable time-step integration routine in the manipulator simulation module, the manipulator input must be updated at unequal time intervals. Furthermore, the inputs are derived from the neural net and the servo controller, so the signals throughout the entire feedback loop need to be recomputed at intervals dictated by the manipulator simulation module. The NeuralWorks package is not well suited for this situation, in which it would have to behave as a slave, with the User I/O dictating the loop timing.

To circumvent this difficulty, we have written a software module to simulate the backpropagation net, thus eliminating the need for the NeuralWorks package. Our software module is not capable of learning, but it can read the configuration file for any backpropagation net previously trained by NeuralWare, and then simulate that net's feedforward response. This software module is compiled and linked together with the manipulator simulation and servo-controller modules, thereby avoiding the overhead of interprocess communications with NeuralWorks.

5
SIMULATION RESULTS FOR A SINGLE-LINK MANIPULATOR

To evaluate the effectiveness of off-line training and the closed-loop control strategies developed in the preceding two chapters, we have simulated the application of these strategies to a 1 degree-of-freedom flexible-joint manipulator. Although the manipulator itself is relatively simple, the simulation process is far from straightforward. Starting from a primarily trial-and-error approach (for lack of analytical guiding principles), we proceeded to train one network after another. After each training session, one or more system or training parameters were adjusted, based only on our observations of previous training results, and the next training session was initiated. Some of the simulation results actually led to refinements in the training strategy, such as the introduction of net input scaling and the allocation of lower learning rates to the higher network layers. In fact, simulation turned out to be far more than a "proof of concept," but was actually an integral part of the design process itself.

Rather than taking the reader through a chronological presentation of this laborious process, we will review the simulation results in reverse order by first discussing the performance of the closed-loop system, and then returning to some of the more interesting effects observed while varying certain system parameters. Section 5.1 provides a detailed description of the manipulator model used in the simulations, and Section 5.2 presents the closed-loop system behavior when a trained 3–layer backpropagation net is used to control this manipulator. Section 5.3 discusses the observed characteristics of pseudo-random off-line training, and Section 5.4 compares the closed-loop performance of a few different net topologies with that of a mathematical model based controller. In Section 5.5, we compare the results obtained when using training signals with narrower bandwidth. The remainder of the chapter explores the effects of variations in the closed-loop servo gains and the joint flexibility coefficient μ.

5.1 Description of the Robot

As an example, we have chosen a single-link manipulator with joint flexibility. This is the same model as in the example studied by Spong *et al.* [7] using an integral manifold approach based on knowledge of the dynamic equations. The manipulator, shown in Figure 5.1 consists of a motor which is elastically coupled to a uniform thin bar of length l, mass m, and moment of inertia $(1/3)ml^2$. The joint is modeled as a linear torsional spring with stiffness $1/\mu$. The equations of motion are:

$$\frac{ml^2}{3}\ddot{q} + B_l\dot{q} + \frac{mgl}{2}\sin q + z = 0 \tag{5.1}$$

$$J_m\ddot{q}_m + B_m\dot{q}_m + \frac{1}{n}z = u \tag{5.2}$$

where

$$z = \frac{1}{\mu}(q - q_j) = \frac{1}{\mu}\left(q + \frac{1}{n}q_m\right)$$

Rewriting equations (5.1) and (5.2) in terms of q and z, we get

$$\ddot{q} = a_1\dot{q} + a_2\sin q + A_1 z \tag{5.3}$$

$$\mu\ddot{z} = a_3\dot{q} + a_2\sin q + a_4\mu\dot{z} + A_2 z + B_2 u \tag{5.4}$$

where

$$a_1 = \frac{-3B_l}{ml^2}, \quad a_2 = \frac{-3g}{2l}, \quad a_3 = \frac{B_m}{J_m} - \frac{3B_l}{ml^2}, \quad a_4 = \frac{-B_m}{J_m},$$
$$A_1 = \frac{-3}{ml^2}, \quad A_2 = A_1 - \frac{1}{n^2 J_m}, \quad B_2 = \frac{1}{nJ_m}.$$

The manipulator parameters used throughout our simulations are listed in Table 5.1.

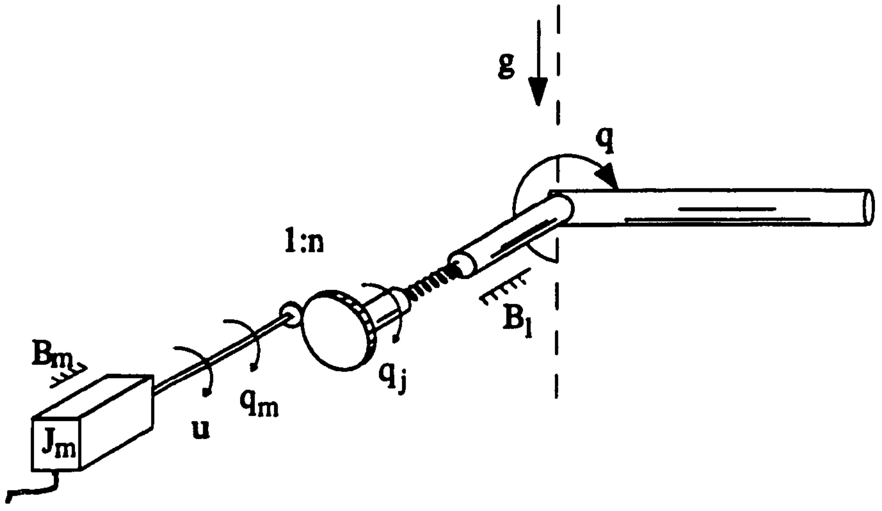


Figure 5.1: Single-link manipulator with joint flexibility.

u	0.001	(1/Nm)
m	10.0	(kg)
l	3.0	(m)
B_m	0.015	(Nms)
B_l	36.0	(Nms)
J_m	0.04	(Nms^2)
n	100	
g	9.8	(m/s^2)

Table 5.1: Manipulator parameters used for simulation.

This system contains the hidden variable z, corresponding to the internal joint torque, which is not observable from the plant outputs $q, \dot{q}, \ddot{q}$ alone. Upon examination of equations (5.3) and (5.4), we see that the system has four state variables, of which two correspond to the q variable, and the other two to the z variable. We assume a time scale separation between the slow q variable and the fast z variable. It is then possible to decouple the equations and to linearize them about the operating point ($q = \dot{q} = \ddot{q} = 0$) corresponding to plant equilibrium. The eigenvalues of the resulting system are computed using the parameters from Table 5.1, yielding approximate pole locations $-0.22 \pm 0.54j$ and $-0.19 \pm 5.98j$, as illustrated in Figure 5.2. The time scale separation between the two complex-conjugate pairs is by a factor of 10, thus supporting our original assumption. The pole pair at $-0.19 \pm 5.98j$ corresponds to the fast z variable of the system. The hidden variable is therefore maximally excited at a frequency of $5.98/2\pi = 0.95Hz$. At frequencies in the vicinity of $0.95Hz$ and higher, the error due to the unobservable variable z becomes unacceptably large.

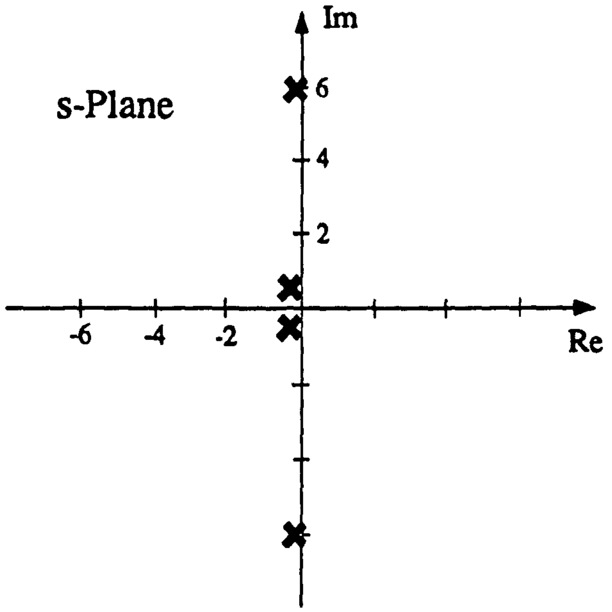


Figure 5.2: Poles of flexible joint used in simulation.

Solving equation (5.3) for z and substituting into equation (5.4), we get:

$$\begin{aligned} u &= f\left\{q, \dot{q}, \ddot{q}, q^{(3)}, q^{(4)}\right\} \\ &= \frac{1}{A_1 B_2}\left[(a_1 A_2 - a_3 A_1)\,\dot{q} + (a_1 a_4 \mu - A_2)\,\ddot{q} - \mu\,(a_1 + a_4)\,q^{(3)}\right. \\ &\qquad + (\mu)\,q^{(4)} + a_2\,(A_2 - A_1)\sin q + (a_2 a_4 \mu)\cos q \cdot \dot{q} \end{aligned}$$

$$-(a_2\mu)\cos q \cdot \ddot{q} + (a_2\mu)\sin q \cdot \dot{q}^2] \tag{5.5}$$

From equation (5.5), we see that the input motor torque is a continuous nonlinear function of the link's joint position q and its first four derivatives. In state-space form, we can write

$$\dot{\mathbf{x}} = g(\mathbf{x}, \mathbf{u})$$

where g is a nonlinear function of the state vector $\mathbf{x} = \left[q, \dot{q}, \ddot{q}, q^{(3)}\right]^T$, and the input vector $\mathbf{u} = u$. This system is observable only from the four states $q, \dot{q}, \ddot{q}, q^{(3)}$; however for model identification we also have to take $q^{(4)}$ as an output. Monitoring (or approximating) the system outputs $q, \dot{q}, \ddot{q}, q^{(3)}, q^{(4)}$ would therefore result in a system which could be used for training a neural net to any desired degree of accuracy.

The preceding analysis and plant description has been provided soleiy for the information of the reader and to aid with the interpretation of the simulation results. Although the robot simulation module implements the above equations, they were neither used nor was their availability assumed for either neural net training or overall system design.

5.2 Closed-Loop System Response with Trained Net

After approximately 2.5 hours of off-line training, the 3–layer backpropagation net described in Chapter 3 was inserted into the closed-loop system. The servo gains were set to $k_0 = 40$, $k_1 = 600$, $k_2 = 4000$, and $k_3 = 10000$, corresponding to a quadruple system pole at $s = -10$. We present the closed-loop results for three different trajectories: a low-frequency high-amplitude sinusoid, an exponential step, and a high-frequency low-amplitude sinusoid.

Figures 5.3 – 5.8 illustrate the steady-state system response to the desired trajectory $q_d = 40sin(0.5t)$. Although the frequency of this signal is only $0.08Hz$, its amplitude of 40 *rad.* causes very high peak joint velocities of 20 *rad/sec* (1200°/*sec*). Despite these high speeds, the tracking error in both position and velocity remains below 0.1% of the input signal. However, as higher derivatives are taken, the percentage error increases dramatically, surpassing 10000% in the fourth derivative of joint position $q^{(4)}$. This error signal (Figure5.7) exhibits distinct high-frequency oscillations in a frequency band at least an order of magnitude higher than that of the system input signal. Although when viewed as a percentage of the desired signal $q_d^{(4)}$, the error seems disconcertingly high, it is actually quite reasonable in an absolute sense. In fact, its relative prominence is due primarily to the nature of the desired trajectory itself. Because this particular trajectory is a pure sinusoid of low frequency, the desired value of the second derivative of acceleration is close to zero, so even slight variations in the actual output $q^{(4)}$ appear relatively large. When a desired trajectory of higher-frequency is used, these oscillations in the higher derivatives become relatively insignificant.

From the driving torque in Figure 5.8, it is apparent that the controller is responding to these oscillations by generating a control signal u which also contains this high-frequency component. Furthermore, although these oscillations are present in the higher derivatives of the output, the controller effectively prevents them from affecting the joint position and velocity. Since in most robotics applications the emphasis is on tracking position and velocity (and sometimes acceleration), we conclude on this basis that the overall system response to the applied low-frequency trajectory is acceptable[10]. It can, however, be argued that these high-frequency oscillations are not being sufficiently damped, and that an input which excites the fast modes of the manipulator could cause more severe errors or even make the system response unstable. We will investigate this possibility by looking at a high-frequency input trajectory.

[10] In cases where more emphasis is to be placed on control of the higher derivatives, a more suitable placement of the closed-loop poles could be obtained by adjusting the servo gains.

The last plot in this series shows the difference between the outer-loop feedback input to the net (v), and the fourth derivative of the manipulator's actual position ($q^{(4)}$). Although $q^{(4)}$ is not used at all in the controller, we recall that is was used as the fifth input to the net during training. In fact, the only net input connection that has been changed since then is this fifth input, which is now connected to v. According to the theory developed in Chapters 3 and 4, the limiting case of a "perfectly-trained" net with no time delays should result in $q^{(4)} \approx v$. We therefore use $\left(q^{(4)} - v\right)$ as a measure of the net's modeling error. For this particular trajectory, $\left(q^{(4)} - v\right)$ oscillates within $\pm 1000\ rad/s^4$, as shown in Figure 5.8.

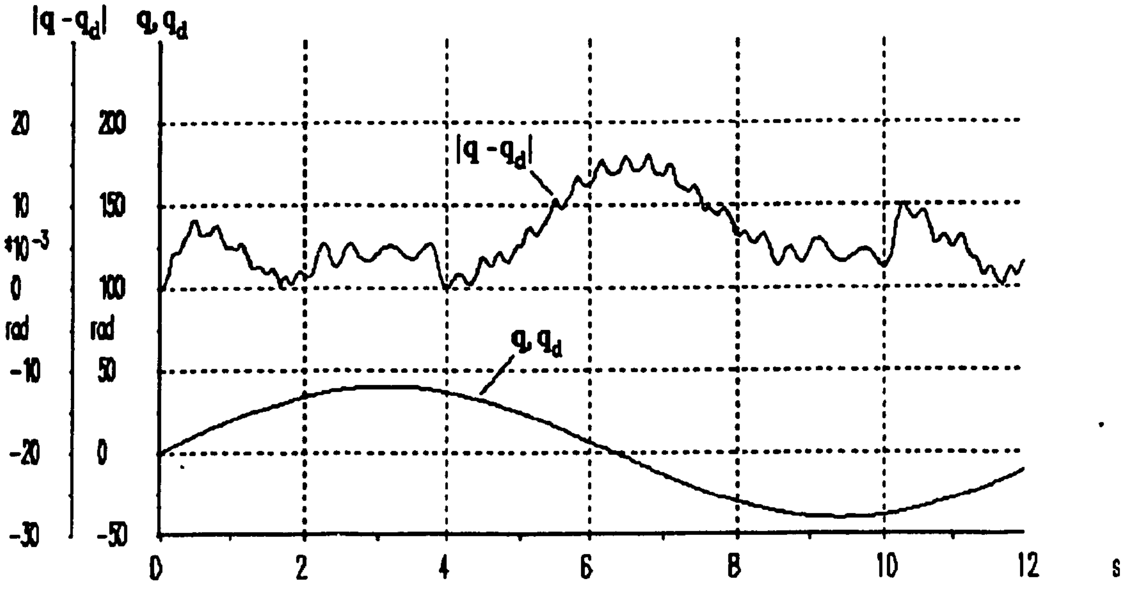


Figure 5.3: Joint position and tracking error at steady state. $q_d = 40\sin(0.5t)$.

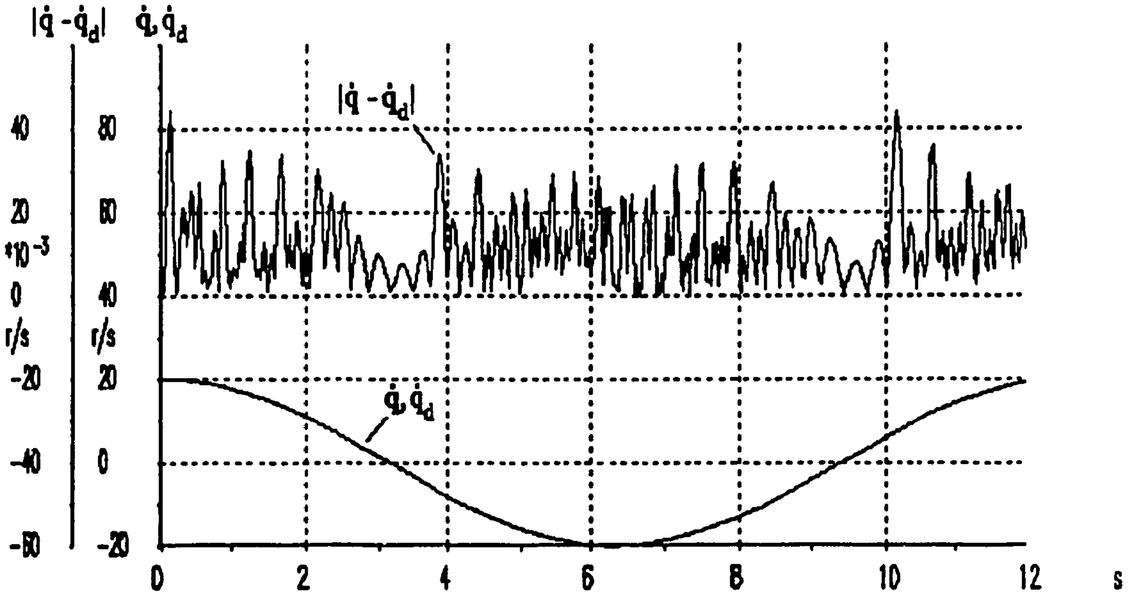


Figure 5.4: Joint velocity and tracking error at steady state. $q_d = 40\sin(0.5t)$.

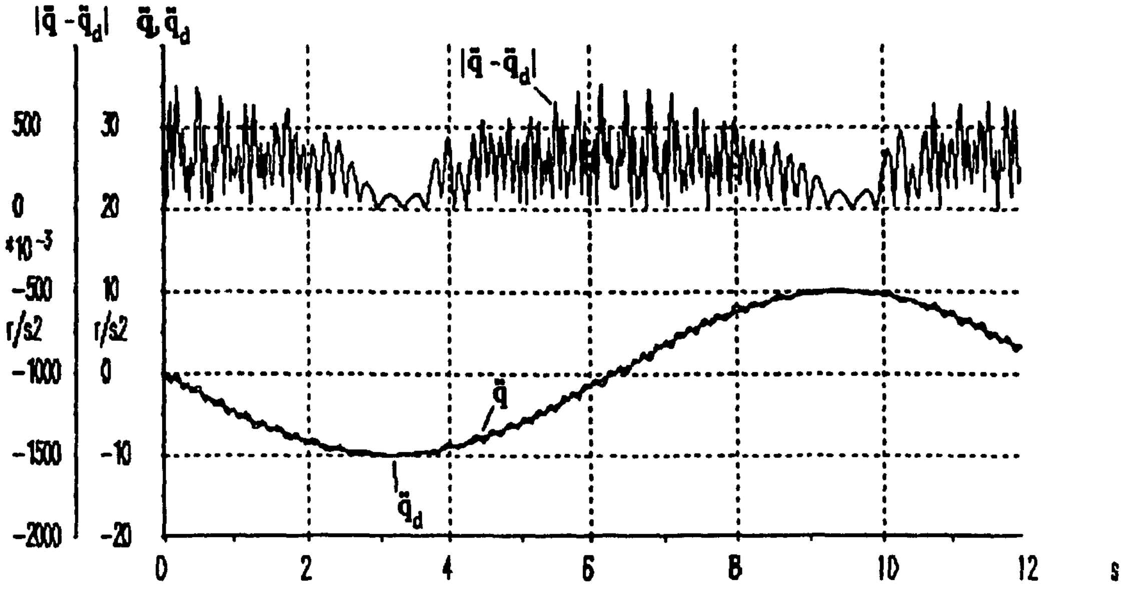


Figure 5.5: Joint acceleration and tracking error at steady state. $q_d = 40\sin(0.5t)$.

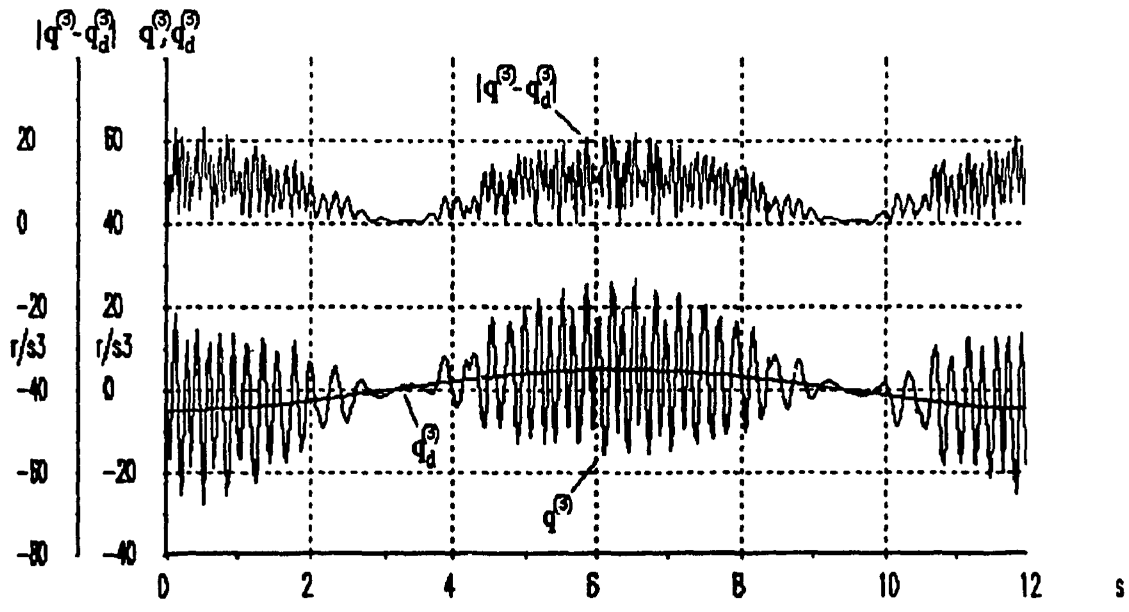


Figure 5.6: First derivative of joint acceleration (jerk) and tracking error at steady state. $q_d = 40\sin(0.5t)$.

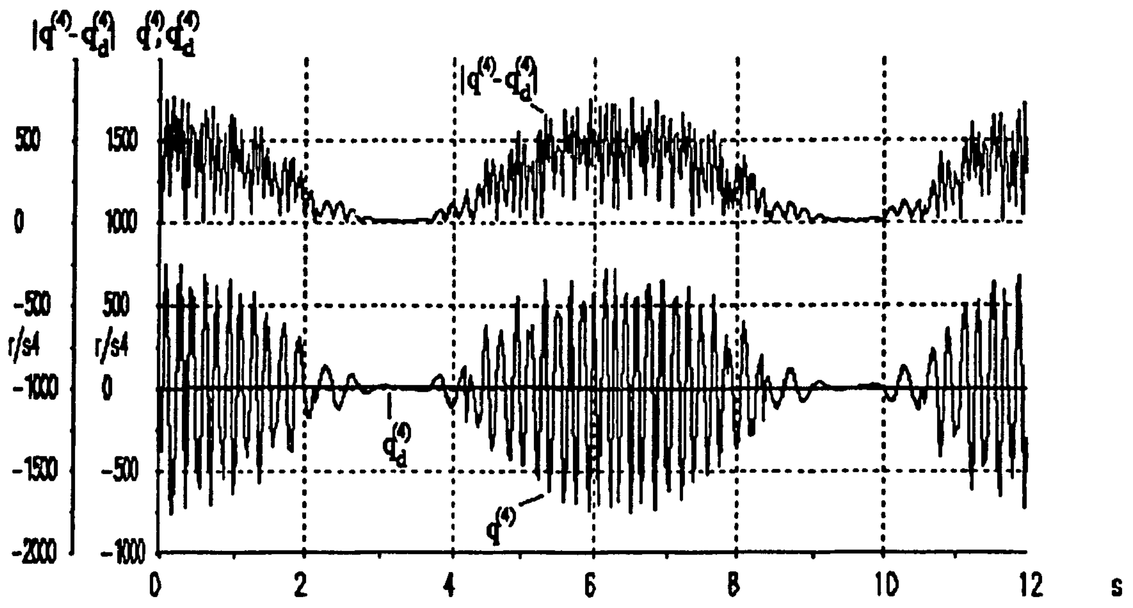


Figure 5.7: Second derivative of joint acceleration and tracking error at steady state. $q_d = 40\sin(0.5t)$.

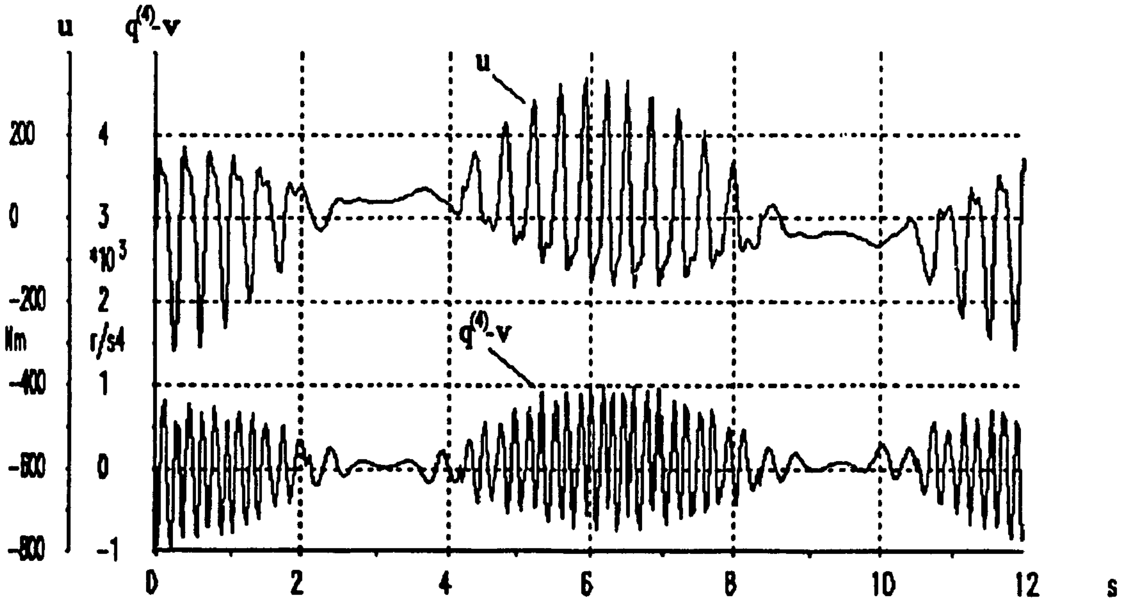


Figure 5.8: Driving torque and modeling error at steady state. $q_d = 40\sin(0.5t)$.

Figures 5.9 – 5.14 illustrate the system's transient response to the desired trajectory $q_d = 1 - e^{-5t}$. At time $t = 0$, the manipulator is at rest with initial conditions: $q_d(0) = \dot{q}_d(0) = \ddot{q}_d(0) = q_d^{(3)}(0) = q_d^{(4)}(0) = 0$. When the desired trajectory is applied at $t = 0$, it creates a discontinuity in the velocity error feedback signal $\dot{e}$ and all of its derivatives. The controller successfully reduces errors in all derivatives to below 5% of their initial values within 1 second.

The high-frequency oscillations observed in the previous trajectory are not present after the first second of this exponential trajectory. A closer examination of $q^{(4)}$ in Figure 5.13 reveals the initial excitation of the fast modes during the first 0.2 *sec*, but these oscillations are very quickly damped as the transient error settles.

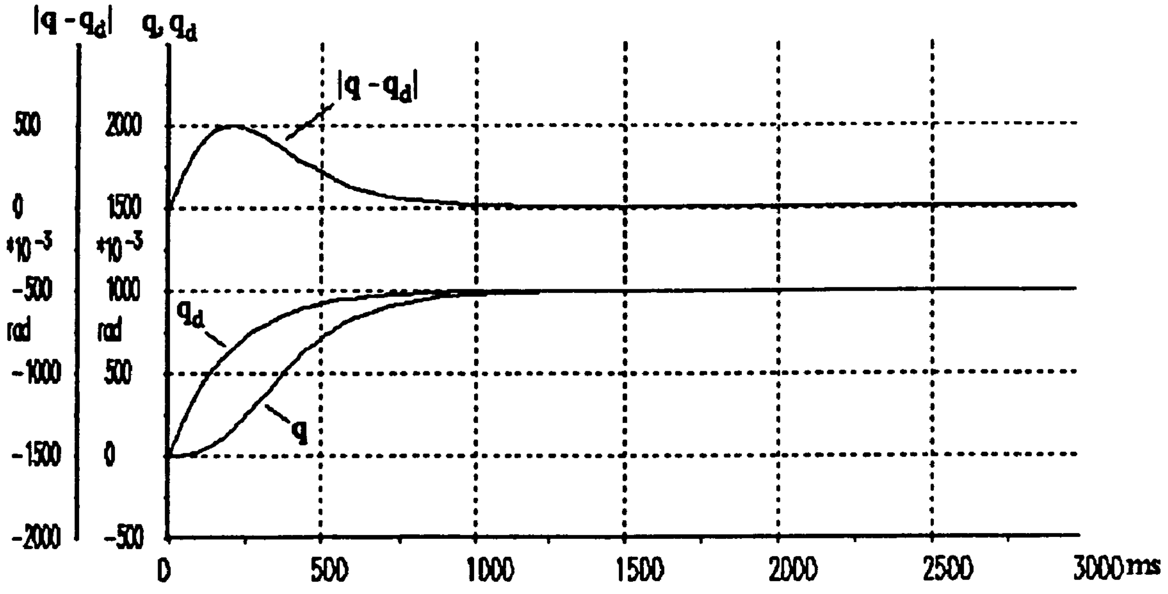


Figure 5.9: Joint position and tracking error during transient. $q_d = 1 - e^{-5t}$.

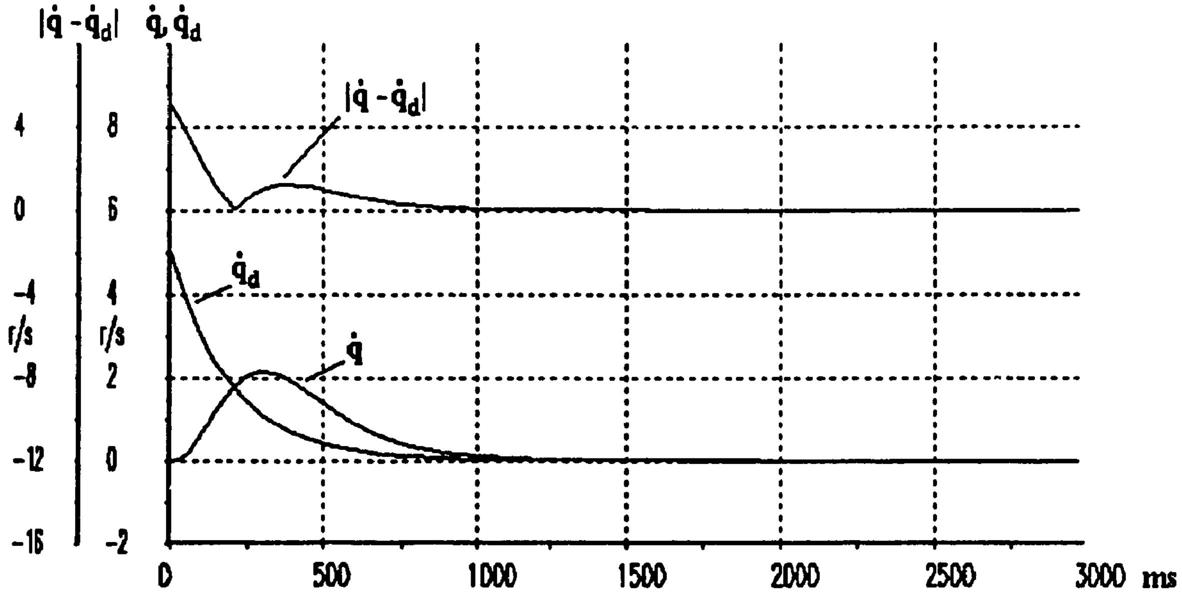


Figure 5.10: Joint velocity and tracking error during transient. $q_d = 1 - e^{-5t}$.

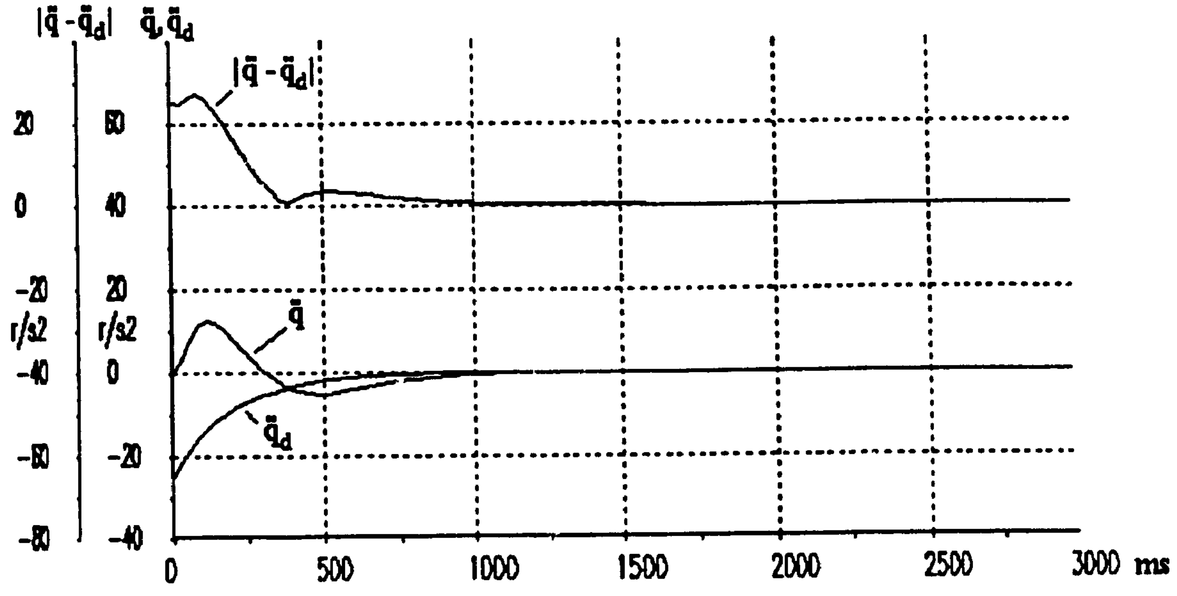


Figure 5.11: Joint acceleration and tracking error during transient. $q_d = 1 - e^{-5t}$.

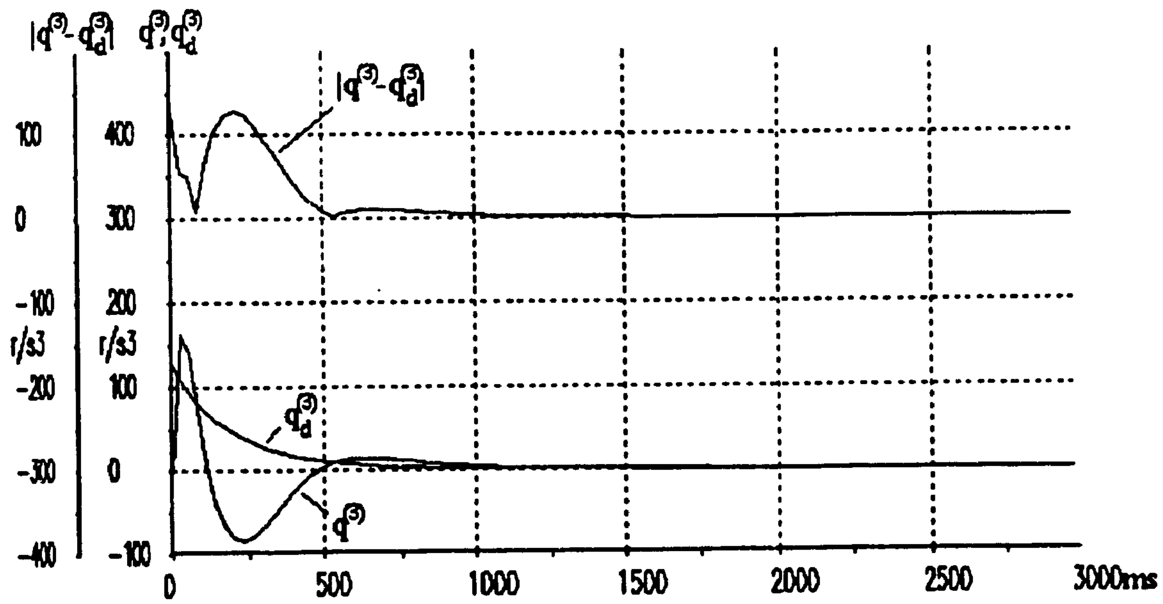


Figure 5.12: First derivative of joint acceleration (jerk) and tracking error during transient. $q_d = 1 - e^{-5t}$.

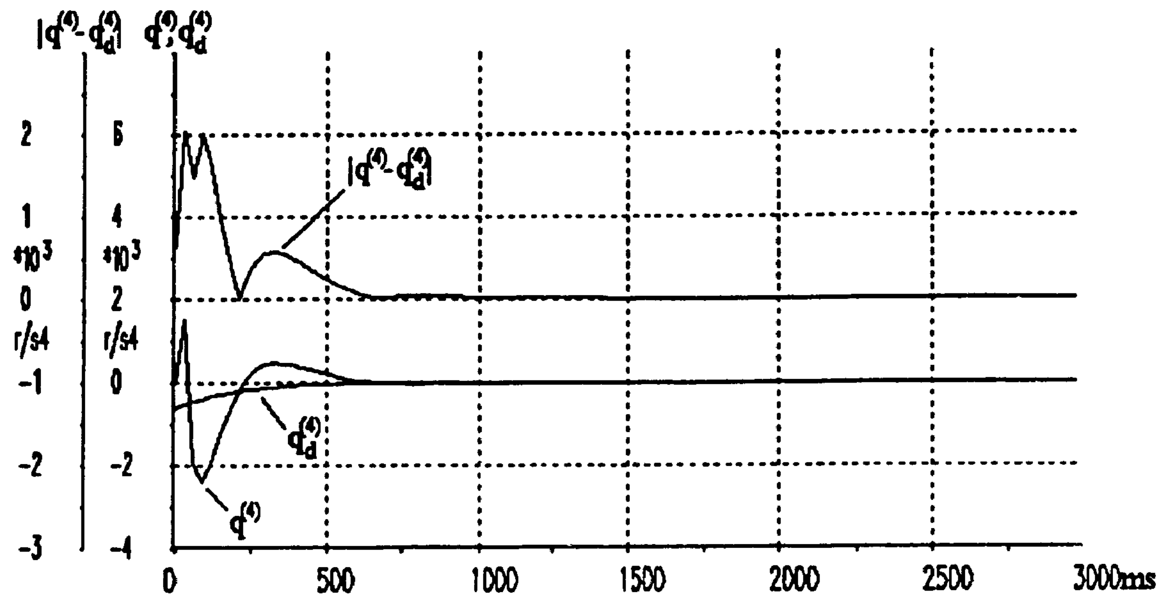


Figure 5.13: Second derivative of joint acceleration and tracking error during transient. $q_d = 1 - e^{-5t}$.

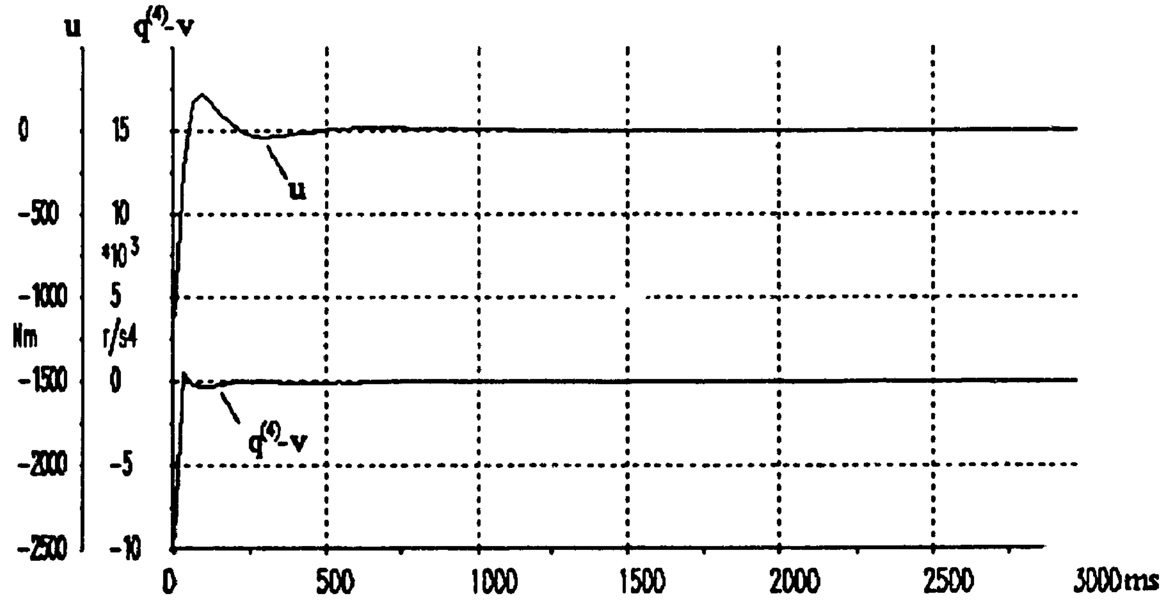


Figure 5.14: Driving torque and modeling error during transient. $q_d = 1 - e^{-5t}$.

Figures 5.15 – 5.20 illustrate the system's response to the high-frequency sinusoidal trajectory $q_d = sin\,(10t)$. By setting all the initial conditions for the manipulator to zero as before, we can view the transient response as it decays towards the sinusoidal steady-state response. As with the exponential trajectory, the transient error decays within the first second to below 5% of the desired input for q_d and all of its derivatives .

In contrast to the low-frequency sinusoidal trajectory, the error between even $q^{(4)}$ and $q_d^{(4)}$ becomes imperceptible as steady state is approached (Figure 5.19). The high-frequency oscillations observed earlier are again present with approximately the same amplitude as before, but because of the much higher amplitude of $q_d^{(4)}$, these oscillations remain below 5% of the system inputs. As before, the modeling error $\left|q^{(4)} - v\right|$ remains roughly within 1000 rad/s^4 despite an approximately four-fold increase in the driving torque (u) over the torque generated by the low-frequency input trajectory. This serves as an indication of the success of generalized training in that the net has been trained with a fairly well balanced error distribution over a wide operating range.

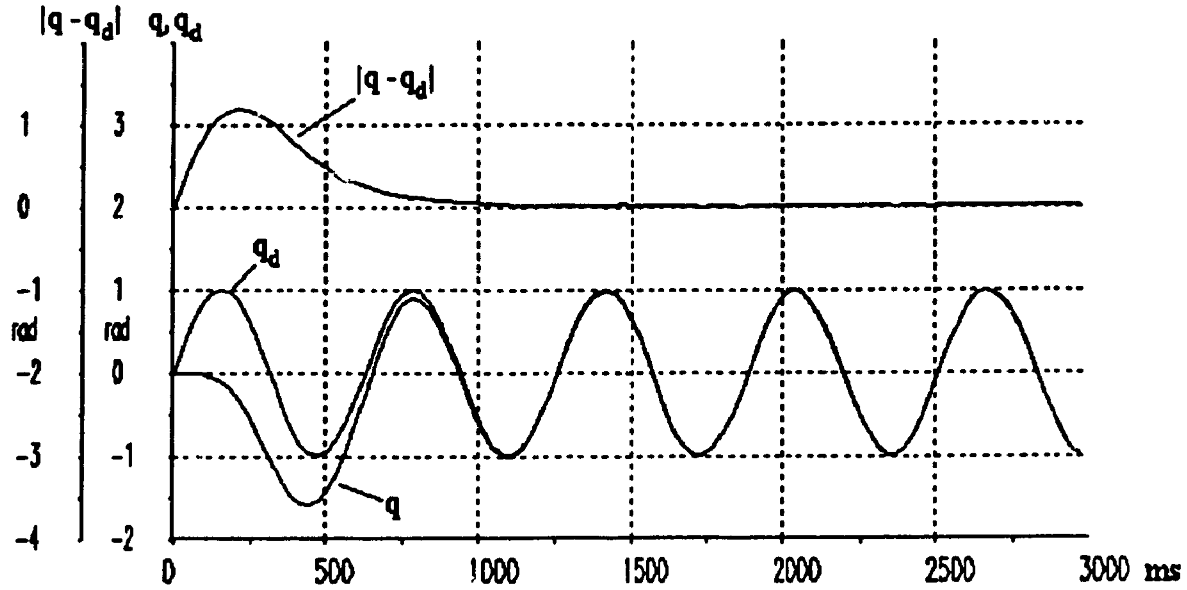


Figure 5.15: Joint position and tracking error at high frequency. $q_d = \sin(10t)$.

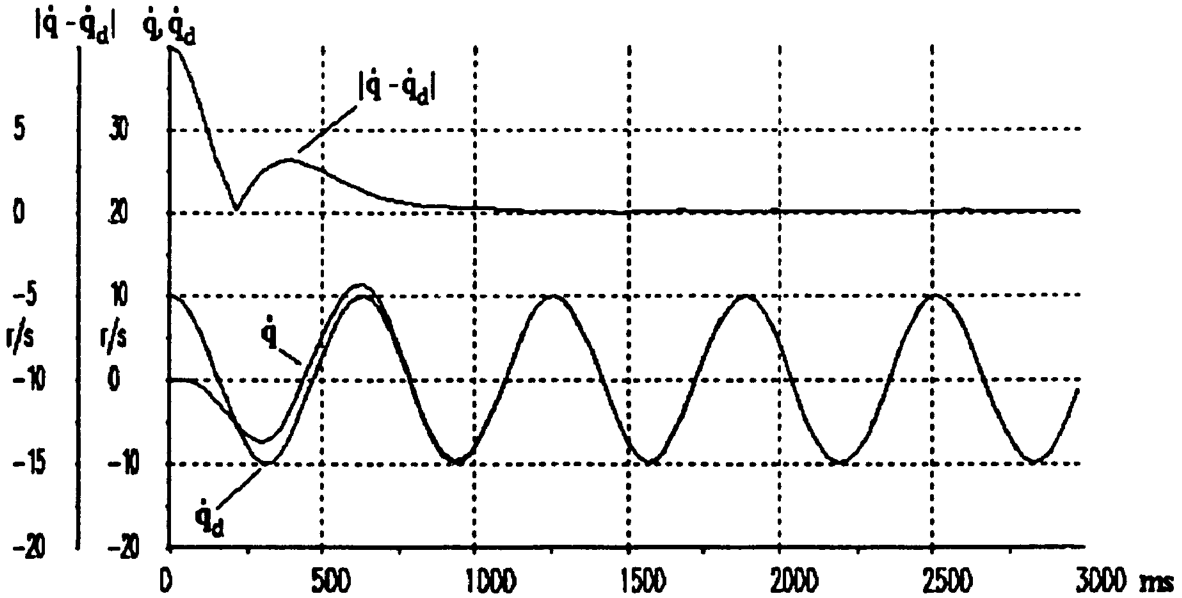


Figure 5.16: Joint velocity and tracking error at high frequency. $q_d = \sin(10t)$.

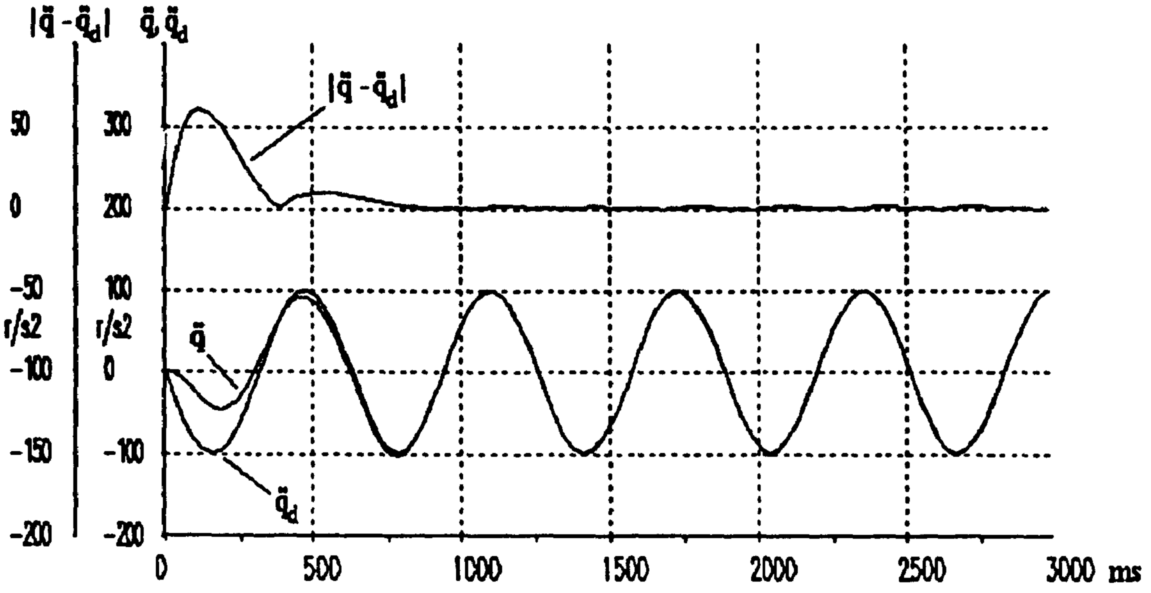


Figure 5.17: Joint acceleration and tracking error at high frequency. $q_d = \sin(10t)$.

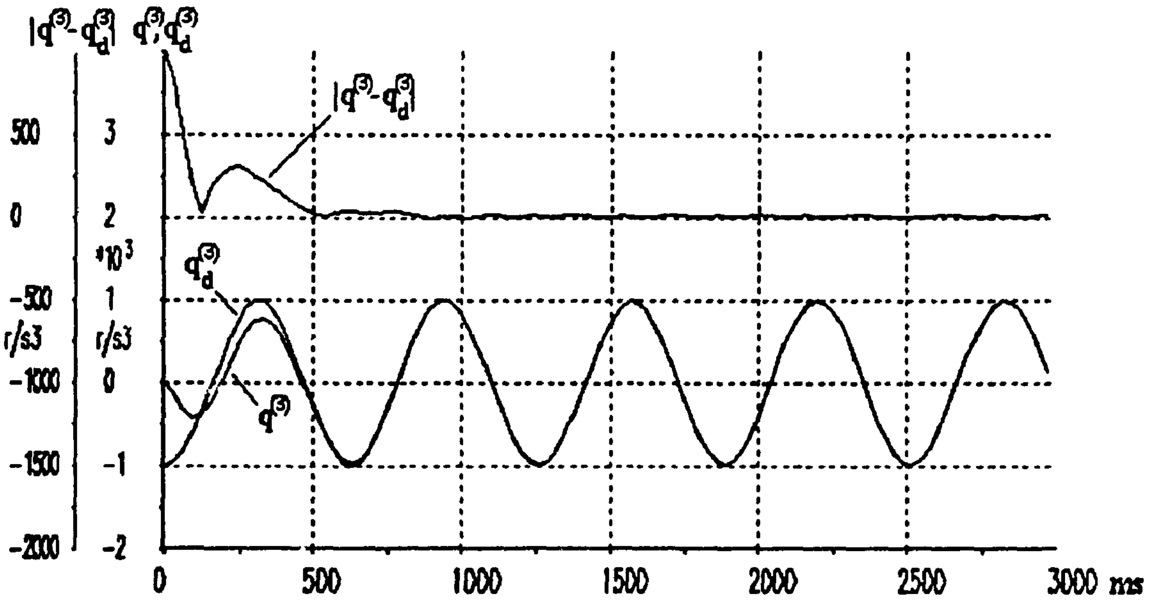


Figure 5.18: First derivative of joint acceleration (jerk) and tracking error at high frequency. $q_d = \sin(10t)$.

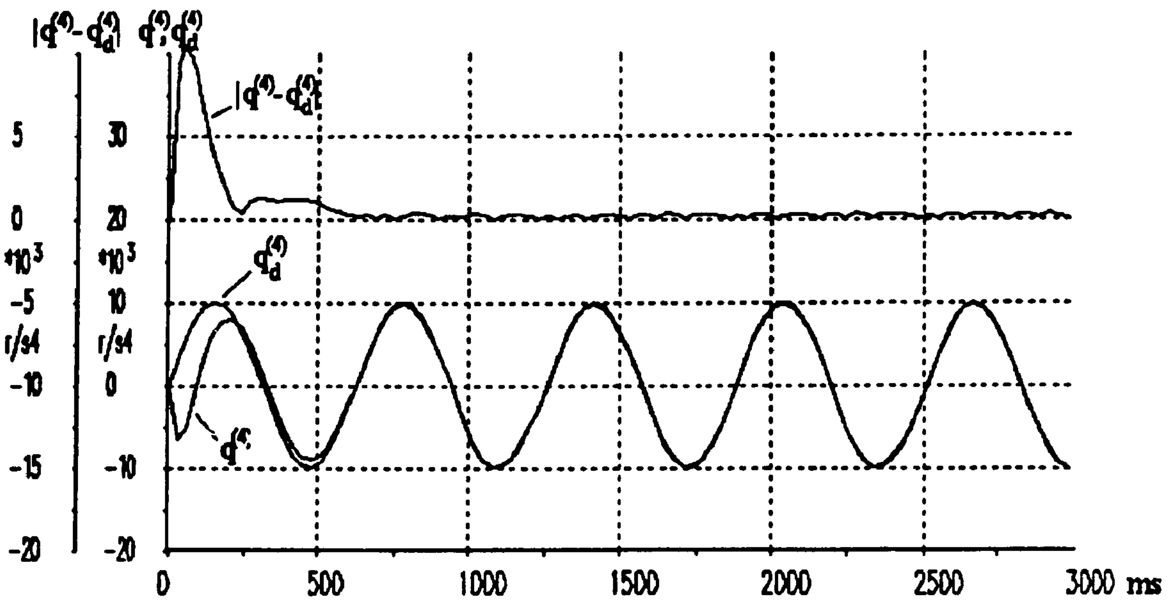


Figure 5.19: Second derivative of joint acceleration and tracking error at high frequency. $q_d = \sin(10t)$.

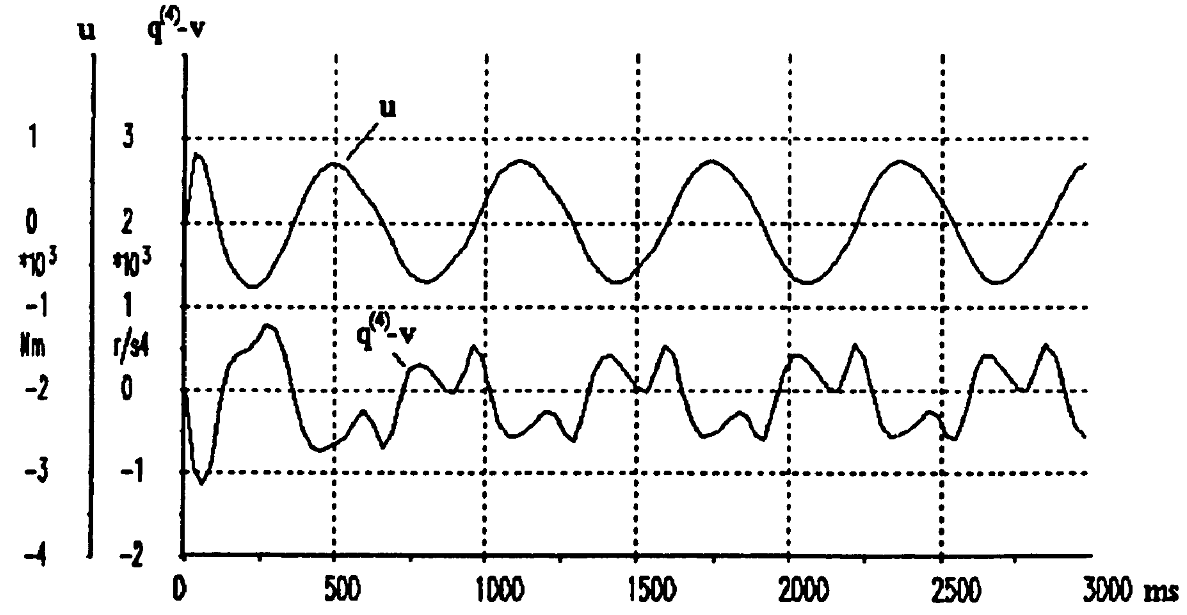


Figure 5.20: Driving torque and modeling error at high frequency. $q_d = \sin(10t)$.

5.3 Off-Line Training

The net used to obtain the closed-loop results described in the previous section was first trained off-line using the lowpass noise training procedure with $A_{max} = 500N$ and $\omega_{max} = 20\pi\ rad/s$ ($10Hz$). Figure 5.21 shows the output error during the first 4 seconds of training. There is a noticeable decrease in the error as the network adjusts its connection weights according to the delta rule of backpropagation. This relatively fast decrease in the error is due primarily to the high initial settings of the learning rate constants. One could expect this sharp downward trend in the error to continue, however we have observed the opposite effect. After falling for a few seconds, the error often increases sharply as training continues, and then decreases again gradually at a similar rate to that shown in Figure 5.21. Such increases in the error are due to the pseudo-random nature of the training signal. If the training signal were just one trajectory being presented repeatedly, we would expect the error (averaged over each presentation of the trajectory) to continue decreasing. A gradual flattening in the error plot would then signal the approach of learning convergence, and serve as a signal to terminate off-line training.

Unfortunately, the results of pseudo-random noise training are much more difficult to interpret. Even after 10000 sec of training, frequent fluctuations in the error are observed (Figure 5.22), and the suitability of the net to any particular trajectory ceases to be a function of training time. For example, we have found cases where a net trained for 9000 seconds performed better in the closed loop system than that same net after it had been trained off-line for an additional 1000 seconds.[11] To minimize such training fluctuations, we implement simulated annealing by reducing the learning rates gradually as a function of time so that the final network weights represent a good "average" model of the inverse plant dynamics. It is toward this end that we train off-line for as long as

[11] In both cases, the desired on-line trajectory was $q_d = \sin(10t)$.

2.5 hours, despite the fact that we have found instances of nets which performed very well on-line after less than one minute of off-line training.

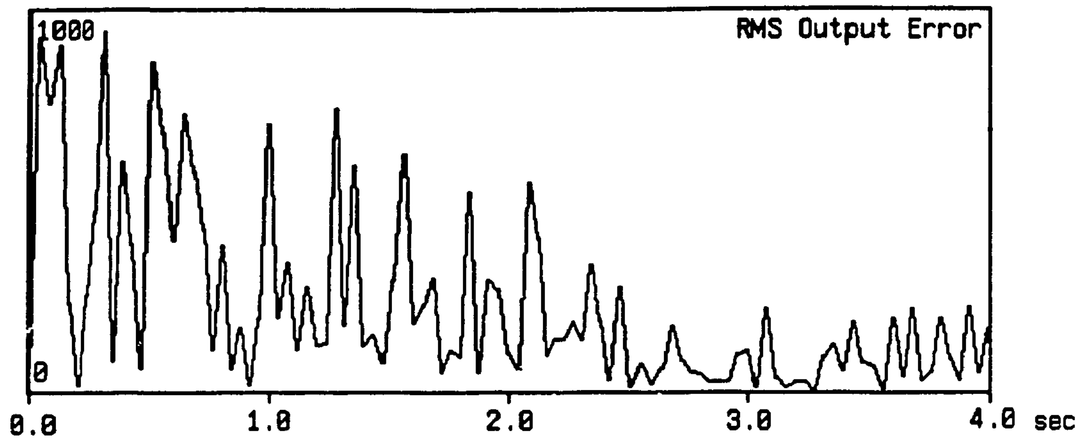


Figure 5.21: Network error during first 4 seconds of training.

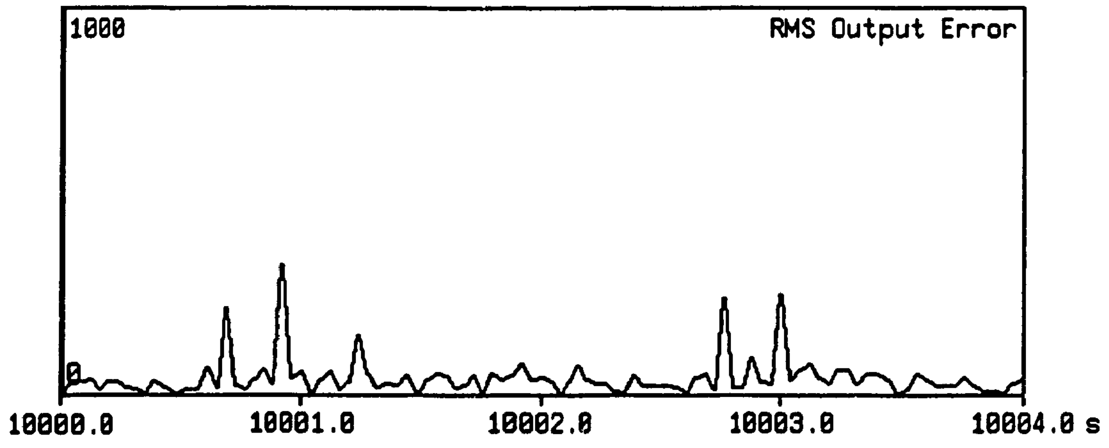


Figure 5.22: Network error near the end of an extended training period.

5.4 A Performance Comparison of Several Nets

With the intention to evaluate the effects of network topology on closed-loop performance, we trained several different nets and tested them on-line. The parameters investigated were the number of layers, the number of nodes, and the activation function of the output node (linear vs. hyperbolic tangent). All nets were trained for 9000 seconds using the simulated annealing procedure described previously. For purposes of comparison, we also ran the simulation with the neural net replaced by the mathematical model of the robot's inverse dynamics, as developed in Section 5.1. The closed-loop results for the mathematical model are shown in Figures 5.23 and 5.24.

Table 5.2 summarizes the closed-loop performance results for the various net topologies in response to the high-frequency desired trajectory $q_d = sin(10t)$. The most striking fact about these results is that the error due to the mathematical model is relatively high. Figure 5.24 shows a modeling error $\left(q^{(4)} - v\right)$ of approximately 300 rad/s^4 peak-to-peak. Although this is only about 1/6th of the modeling error of the 3–layer used in Section 5.2, it is still more than one would expect for a mathematical model. We attribute this error to the numerical integration performed within the robot model. Such discrete integration necessarily introduces some time delays which would directly affect the modeling error as we have defined it.

Although there were variations in the performance of the different nets, we could detect no clear correlation between the performance and the network parameters discussed. The observed differences in performance were actually within the same order of magnitude as the fluctuations caused by pseudo-random training. We therefore conclude that the exact topology is not a critical factor in determining the net's performance, provided that we are dealing with nets having at least the minimum size required to model the dynamics of the manipulator in question.

Another surprising result is that the "Big 4" network's errors in the third and fourth derivatives where actually lower than their mathematical model counterparts. We have observed several such cases in other nets as well, but in all of these instances continued training eventually increased the error. It is possible that such instances of unusually low error are only random effects due to the fluctuations of lowpass noise training. On the other hand, the nets were trained with the same mathematical model of the robot (containing the same integration routine) that is used in the closed-loop system. It is conceivable that the nets actually learned some of the unmodeled dynamics introduced by the discrete simulation. Unfortunately, this result is not consistently reproducible, but it does demonstrate the main strength of the neural network approach: its ability to learn the unmodeled behavior of the system.

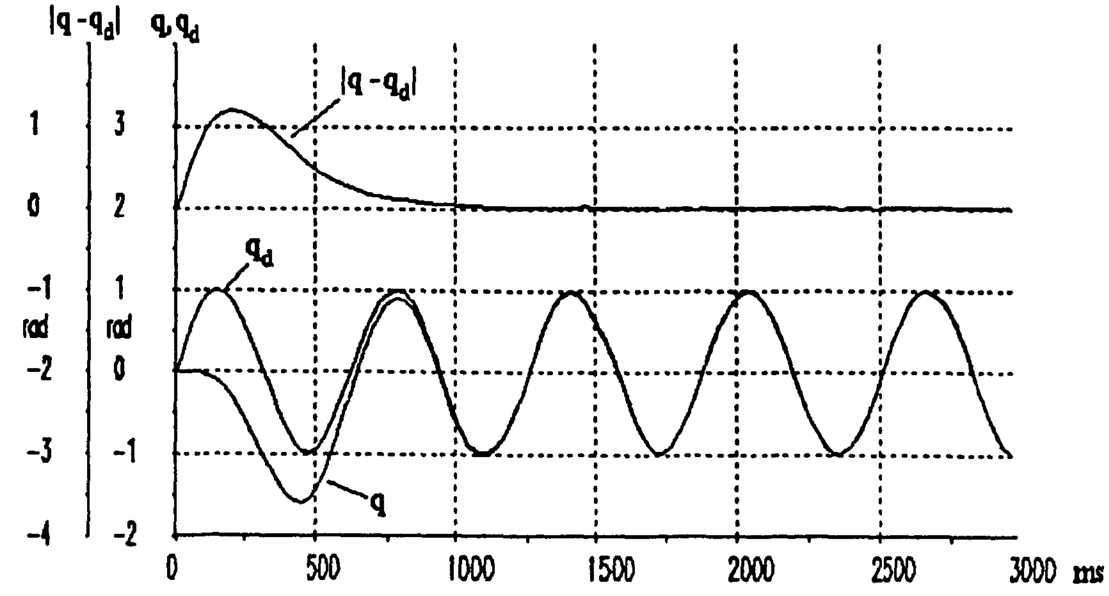


Figure 5.23: Position and tracking error with net replaced by mathematical model.

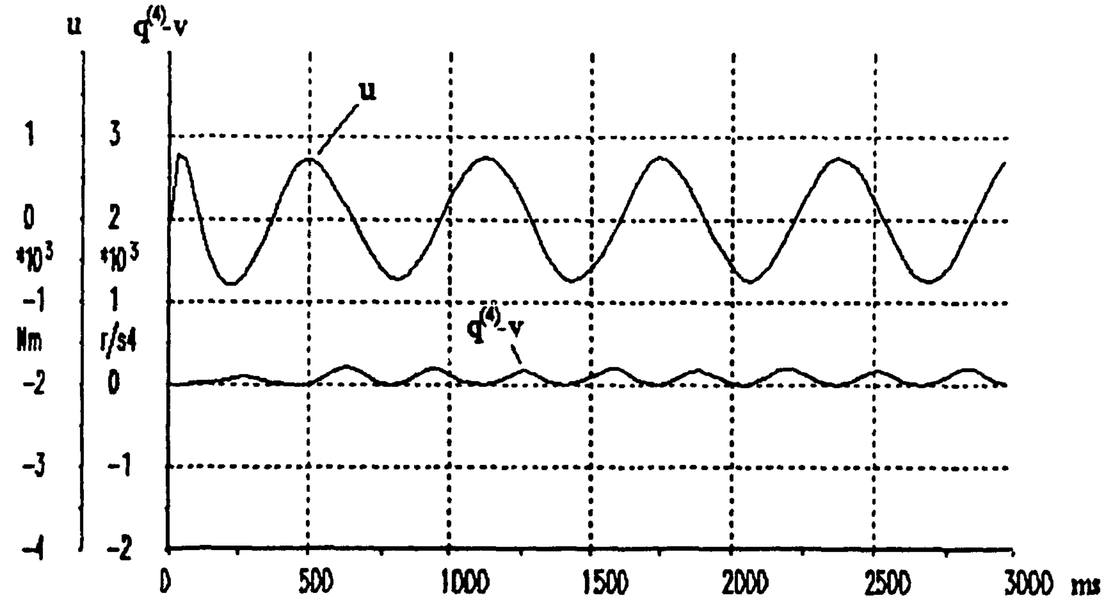


Figure 5.24: Driving torque and modeling error with net replaced by mathematical model.

	Net name:	Tanh 3	Lin 3	Big 3	Tanh 4	Big 4	Math Model
Net Type	Nodes in Hidden Layer 1	20	20	50	20	50	—
	Nodes in Hidden Layer 2	12	12	30	20	30	—
	Nodes in Hidden Layer 3	—	—	—	12	30	—
	Output Node Type	tanh	linear	linear	tanh	tanh	—
Approximate Steady-State RMS Errors	Position (rad)	0.012	0.008	0.008	0.015	0.012	0.008
	Velocity (rad/s)	0.05	0.02	0.05	0.06	0.015	0.006
	Accel. (rad/s-s)	0.6	0.2	0.5	0.6	0.12	0.1
	Jerk (rad/s-s-s)	6.0	3.0	5.0	6.0	1.5	2.0
	Deriv. of Jerk (rad/s-s-s-s)	70.0	30.0	50.0	70.0	30.0	40.0
	Modeling Error (rad/s-s-s-s)	420.0	175.0	385.0	560.0	145.0	140.0

Table 5.2: Average RMS error at steady state for fully-trained nets in closed-loop system. $q_d = \sin(10t)$.

5.5 Changing the Training Signal

To study the effects of varying the lowpass noise training signal, a new three-layer net with the same topology as before was trained, this time with a cut-off frequency of 1/4 of that used to train the original net. With $w_{max} = 5\pi\ rad/s$ (2.5Hz), and $A_{max} = 500N$ (as before), the net was trained until no further significant reductions in average error were observed (9000 sec.). The resulting closed-loop response to the high-frequency desired trajectory is shown in Figure 5.26. The position and velocity error is approximately 500% higher than for the case of the original net (Figure 5.25). The driving torque signal generated by the new net is severely distorted, as shown in Figure 5.30, and the modeling error is more than three times higher than it was for the original net.

These results are not very surprising. One would expect a net trained with a higher frequency signal to perform better for a high frequency trajectory than a net trained within a frequency band that only barely encloses the fundamental frequency of that desired trajectory. Following this line of reasoning, the new net should therefore perform relatively well for our lower-frequency ($0.08Hz$) trajectory which is well within this network's training band. The simulation results in Figure 5.29 show that quite the opposite is true. Compared to the performance of the original net (Figure 5.28), the new net again generates 500% higher position and velocity errors. The modeling error is still three times higher than for the original net. From these results, we can conclude that the closed-loop performance is, indeed, critically dependent in the selection of a suitable training bandwidth, and that this choice is not dictated exclusively by the bandwidth of the on-line trajectories, but is also a function of the manipulator itself.

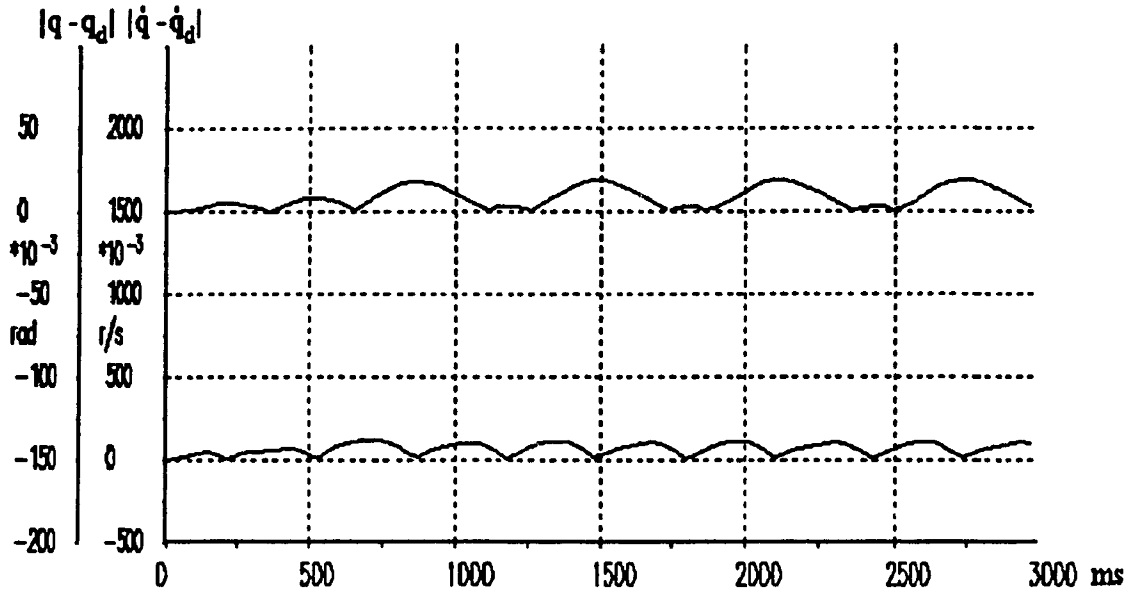


Figure 5.25: Position and velocity tracking errors for original net. $q_d = \sin(10t)$.

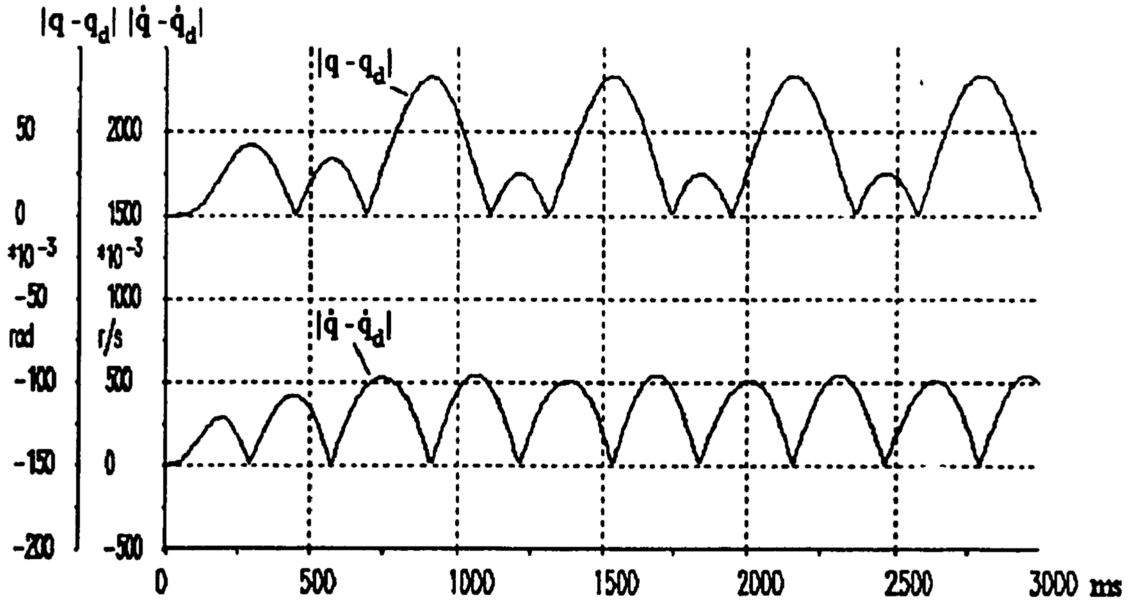


Figure 5.26: Position and velocity tracking errors for net trained with lower-frequency noise. $q_d = \sin(10t)$.

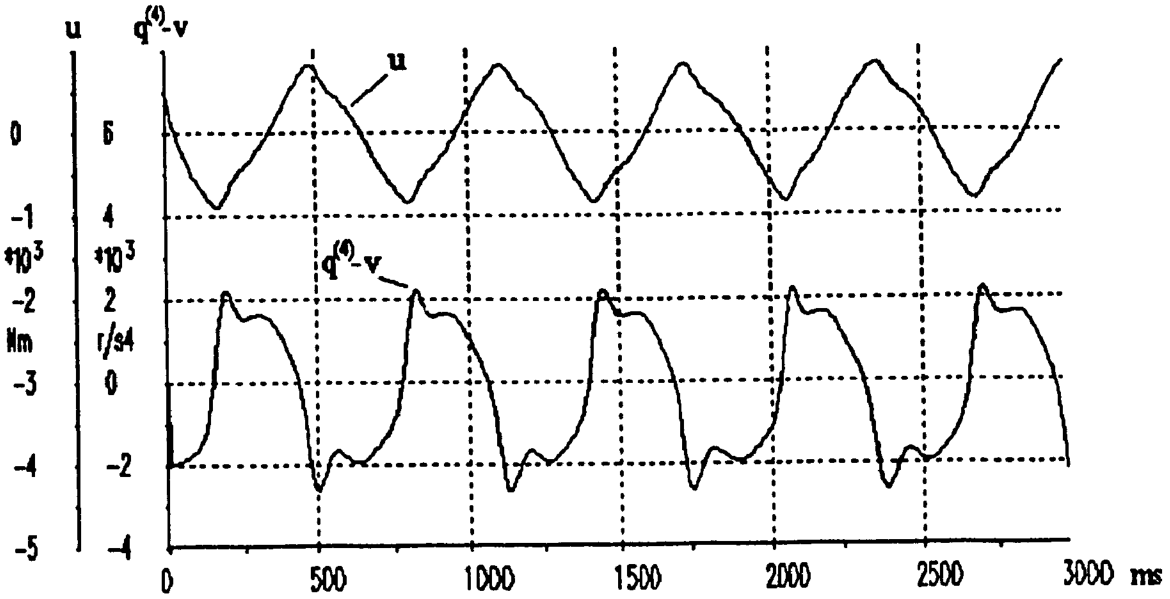


Figure 5.27: Driving torque torque and modeling error for net trained with lower-frequency noise. $q_d = \sin(10t)$.

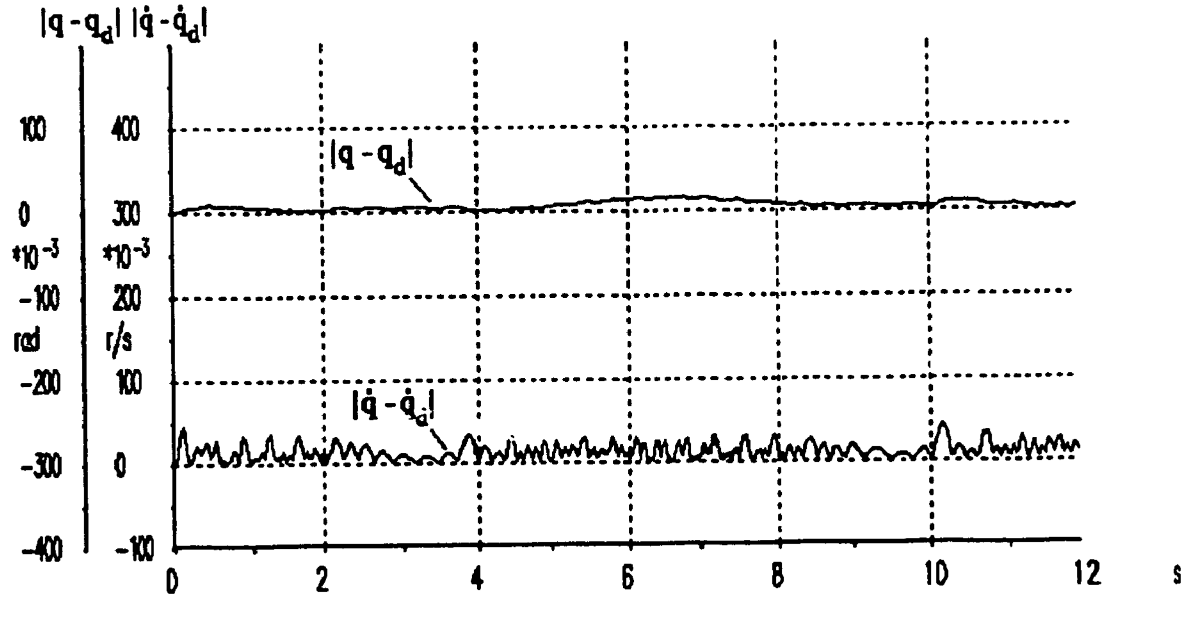


Figure 5.28: Position and velocity tracking errors for original net. $q_d = 40\sin(0.5t)$.

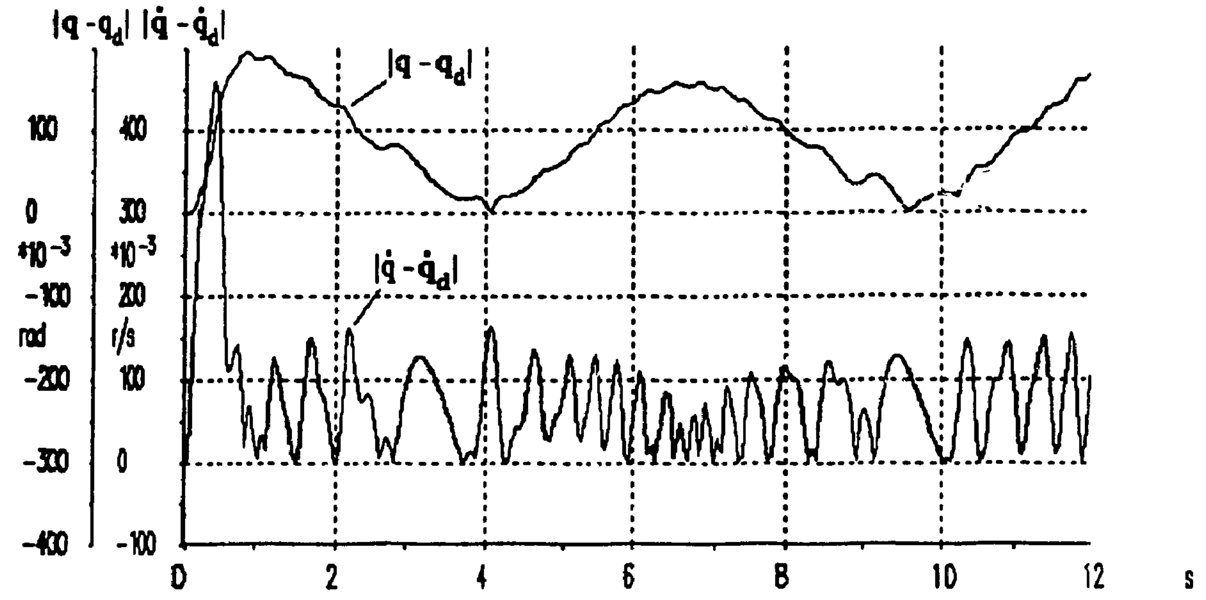


Figure 5.29: Position and velocity tracking errors for net trained with lower-frequency noise. $q_d = 40\sin(0.5t)$.

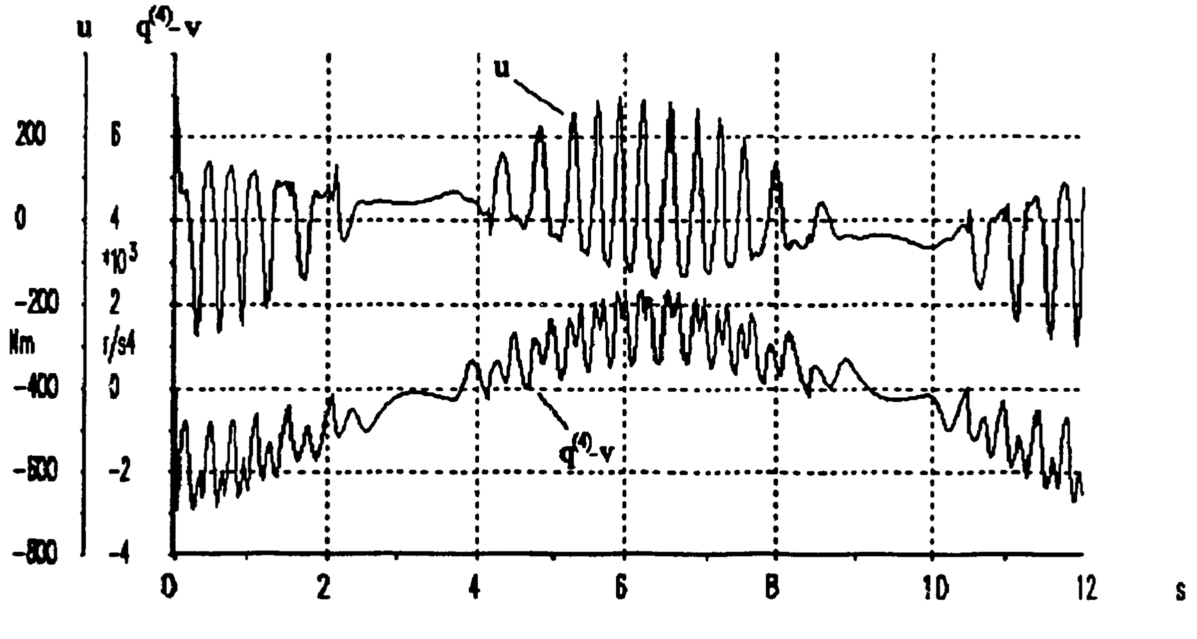


Figure 5.30: Driving torque torque and modeling error for net trained with lower-frequency noise. $q_d = 40\sin(0.5t)$.

5.6 The Effects of Increasing the Servo Gains

Since the controller was designed with the goal that the servo gains alone should define the overall system response, we now attempt to speed up the response by adjusting these gains. Using the same net as in Section 5.2, we move the system poles from $s = -10$ to $s = -15$ by setting the servo gains to $k_0 = 60$, $k_1 = 1350$, $k_2 = 13500$, and $k_3 = 50625$. The corresponding response to the desired input trajectory $q_d = sin(10t)$ is plotted in Figures 5.31 and 5.32. As expected, the transient error decays more quickly, falling to below 5% of its peak within 0.75 *sec* (in contrast to 1.0 *sec* when the poles were at $s = -10$). The faster response does, however, require greater torque u which is now reaching a peak of 1700 N, not far from the actuator limits of 2000 N.

To obtain an even faster response, we place the poles at $s = -20$ by setting the servo gains to $k_0 = 80$, $k_1 = 2400$, $k_2 = 32000$, and $k_3 = 160000$. Rather than further decreasing the response time, however, this adjustment actually makes the system unstable, as shown in Figure 5.33. The driving torque (Figure 5.34) exhibits severe clipping at ± 1850 N, just below the actuator limits. Because we had set the net's output scaling to 2000, and the output node performs the hyperbolic tangent activation function, it is not possible for the net output u to exceed 2000 N. By replacing the net with our mathematical model, we can remove this restriction, and obtained the faster response we desire (Figures 5.35 and 5.36), however the driving torque required to do so reaches 7000 N, clearly violating the actuator limits. The failure of the neurocontroller when the system poles are set at $s = -20$, is therefore not due to a weakness in the control strategy or in the net's modeling accuracy, but to a physical limitation of the manipulator itself. The net actually protects the manipulator by limiting the torque to within the allowed range.

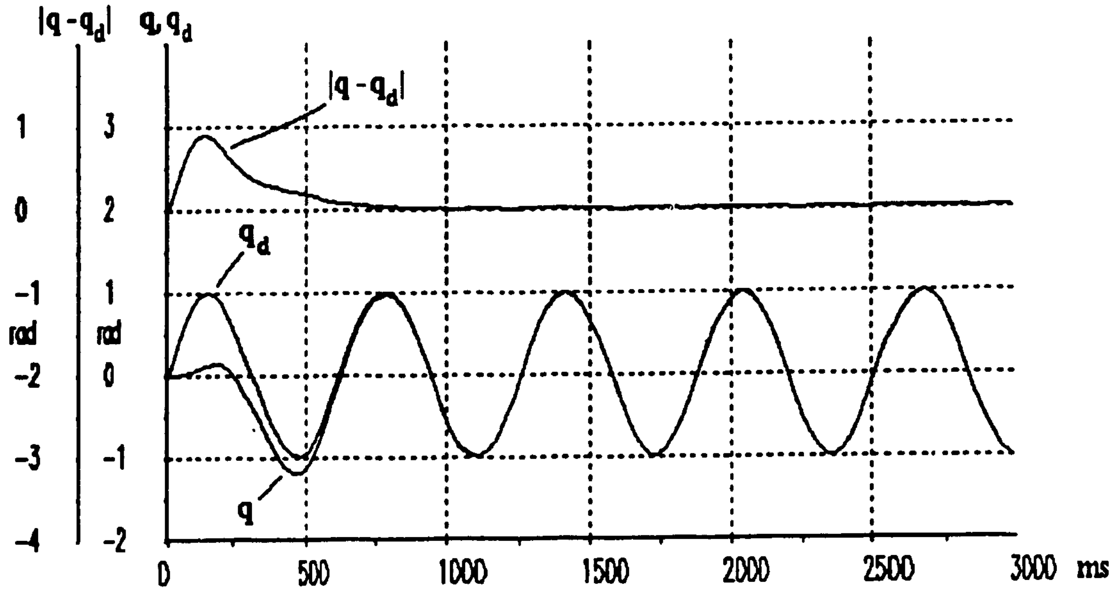


Figure 5.31: Position and tracking error with quadruple pole at s = –15.

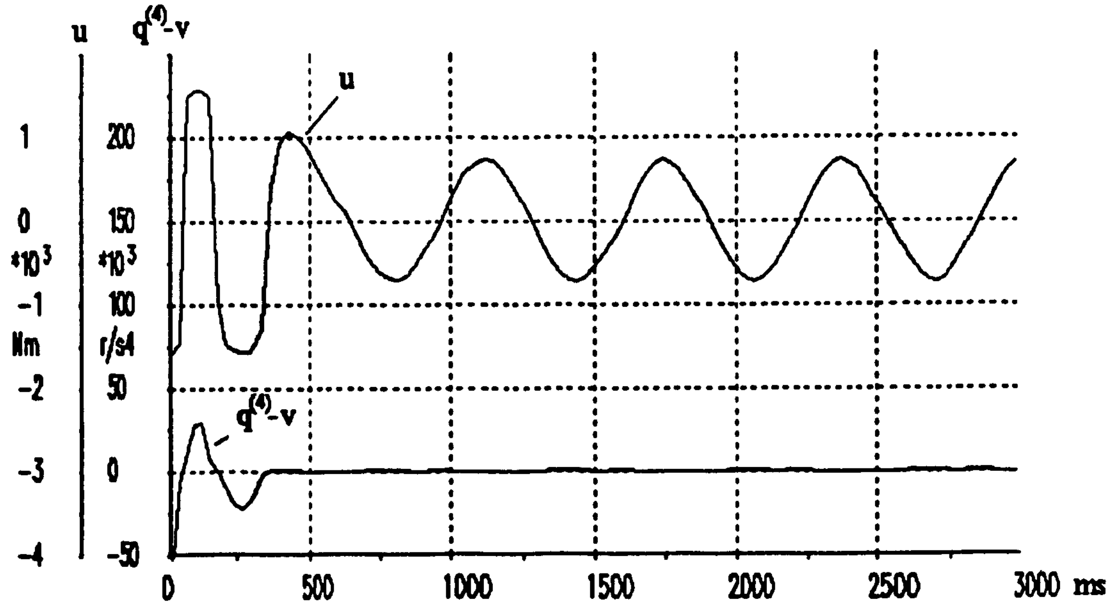


Figure 5.32: Driving torque and modeling error with quadruple pole at s = –15.

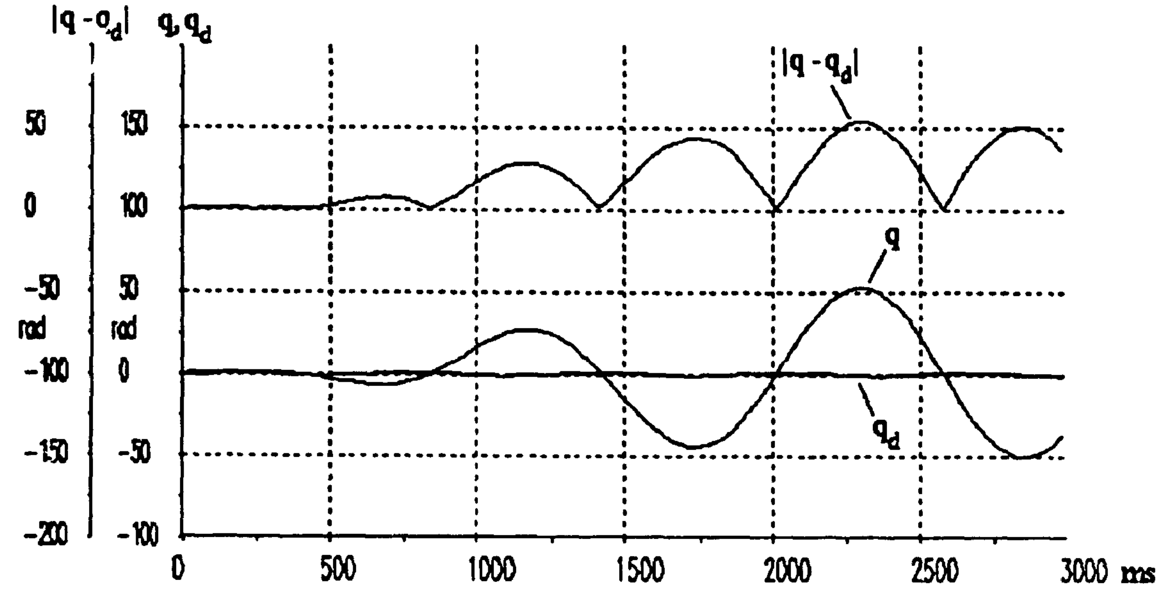


Figure 5.33: Position and tracking error with quadruple pole at s = –20.

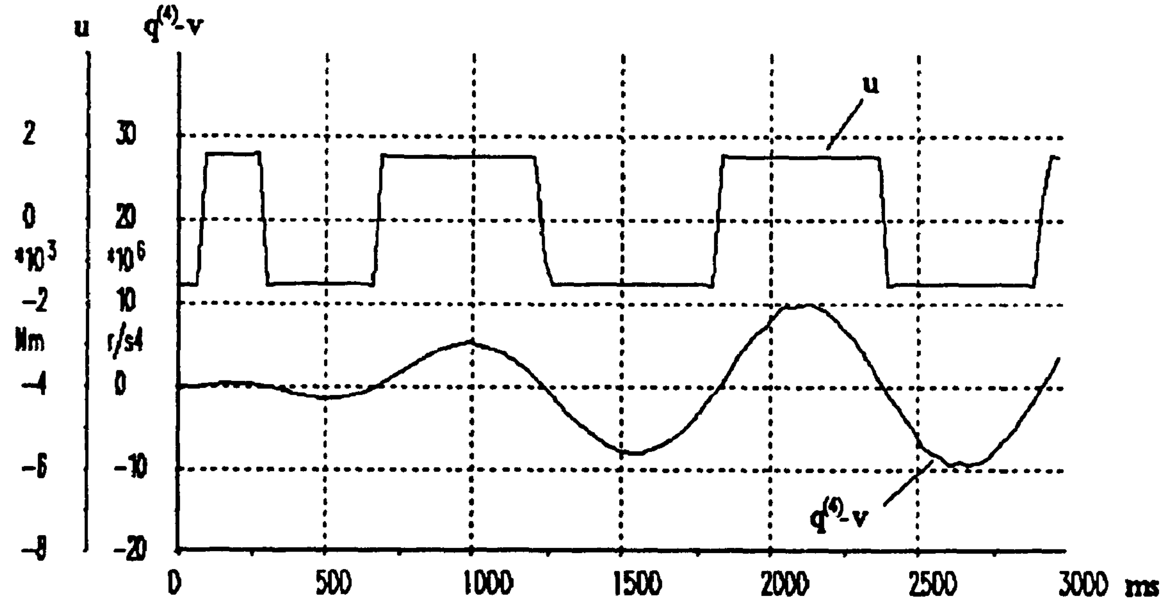


Figure 5.34: Driving torque and modeling error with quadruple pole at s = –20.

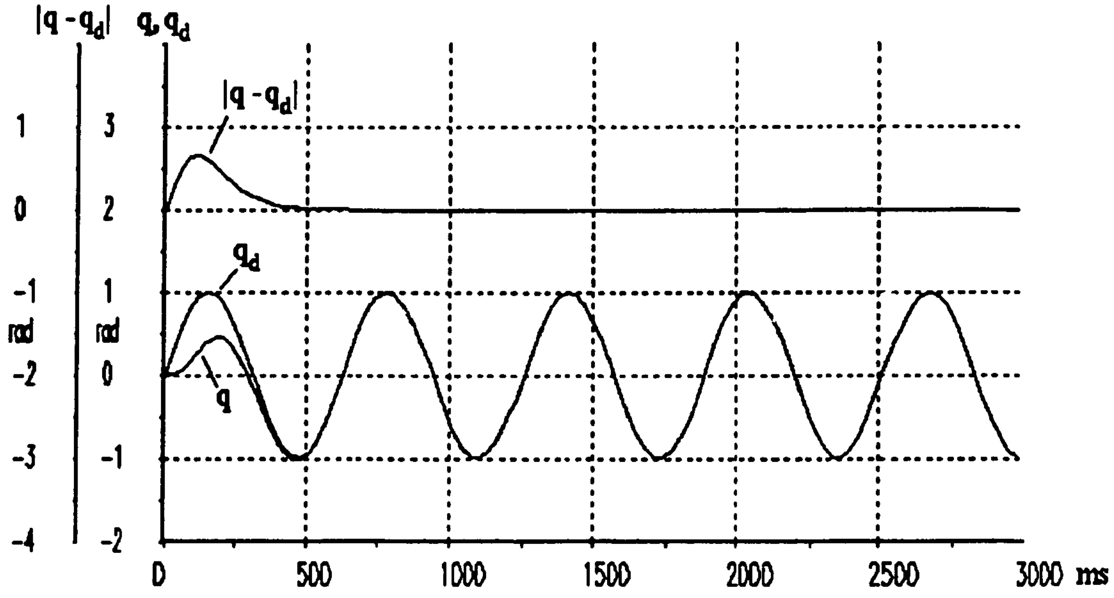


Figure 5.35: Position and tracking error using mathematical model with quadruple pole at s = –20.

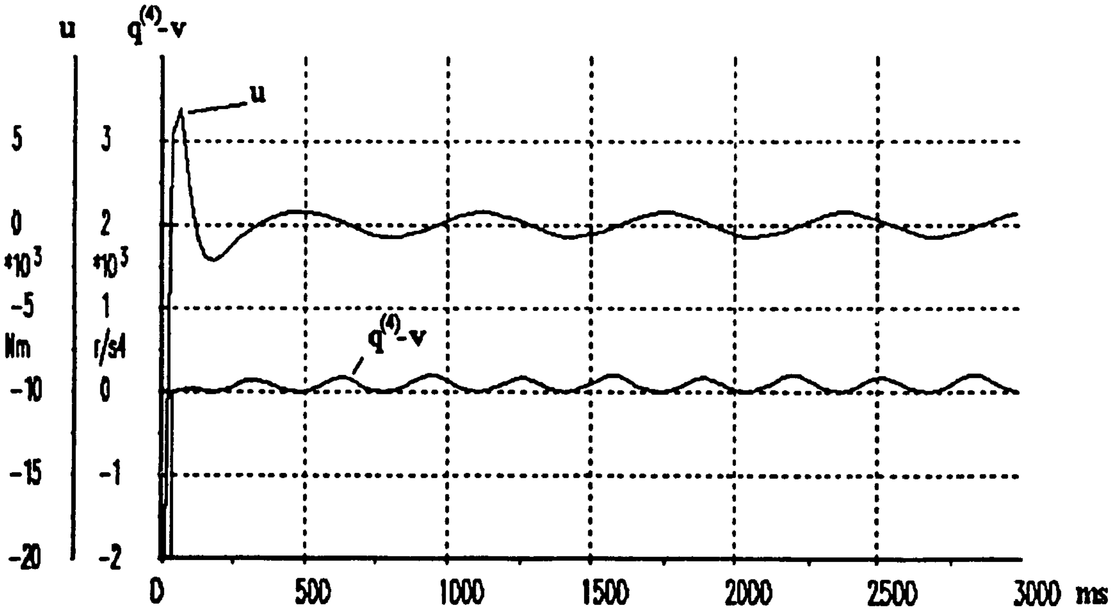


Figure 5.36: Driving torque and modeling error using mathematical model with quadruple pole at s = –20.

5.7 The Effects of Greater Joint Flexibility

Figures 5.37 and 5.38 show the results for a net trained with a manipulator having joint flexibility $\mu = 0.001$ being used in a closed-loop system to control a manipulator with $\mu = 0.01$, and all other parameters remaining unchanged. As could be expected, the increased flexibility results in severe error, and makes the system unstable. Retraining the net off-line with this more flexible manipulator, and then re-installing it in the control system yields stable tracking, as shown in Figures 5.39 and 5.40. However, the driving torque u for this low-frequency desired trajectory is very close to the actuator limits of $\pm 2000\ N$. Any further increases in flexibility, even if the net were re-trained, would tend to cause the torque clipping and the corresponding instability observed in Section 5.6. These effects can be avoided by reducing the servo gains, at the expense of closed-loop performance. In general, although the closed-loop performance can be controlled by adjusting the servo gains as required, such adjustments can only be made within a certain range which is determined by the actuator limits and by the manipulator parameters, including the joint flexibility coefficient.

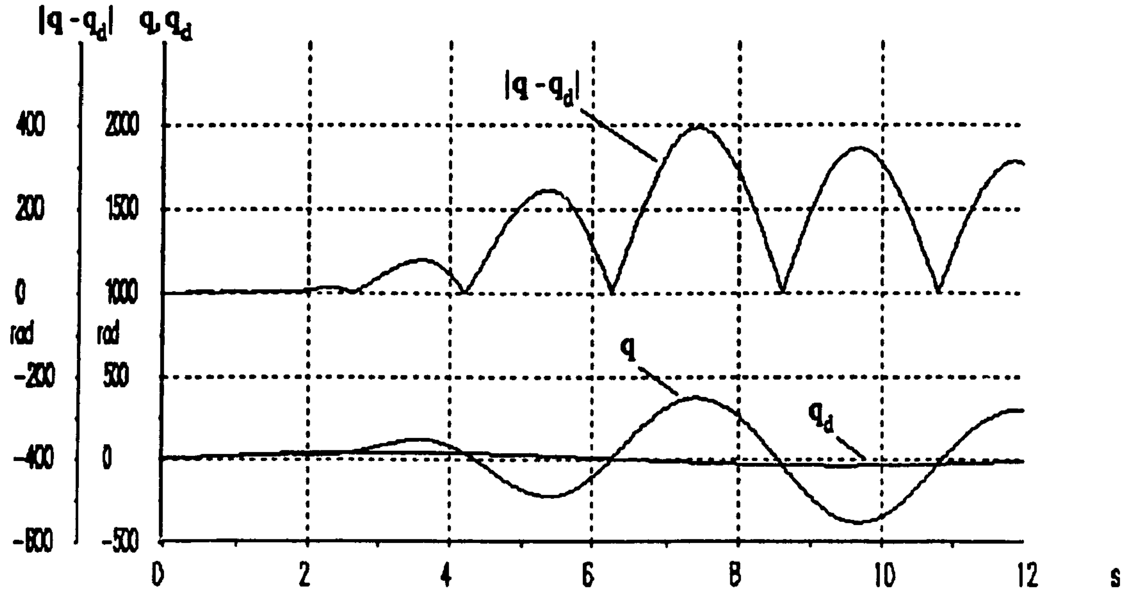


Figure 5.37: Position and tracking error for original net.

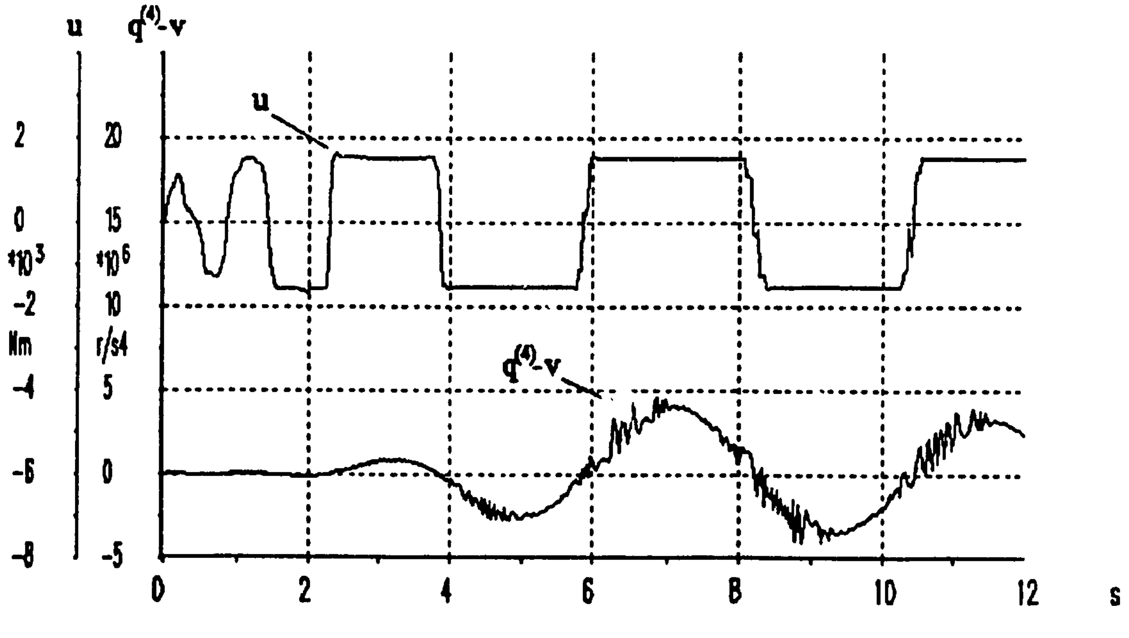


Figure 5.38: Driving torque and modeling error for original net.

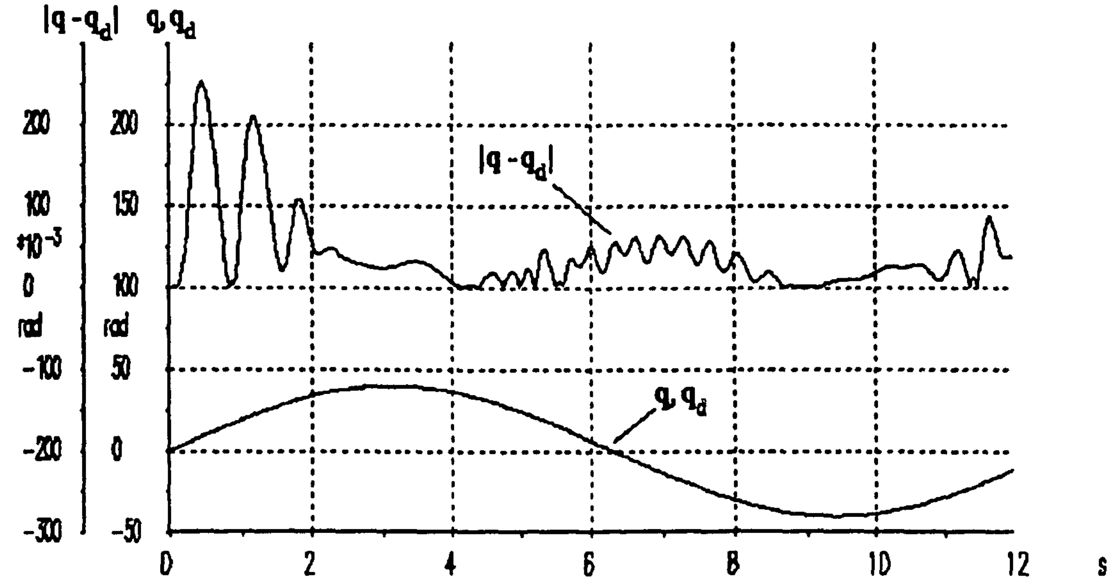


Figure 5.39: Position and tracking error for net trained with more flexible joint.

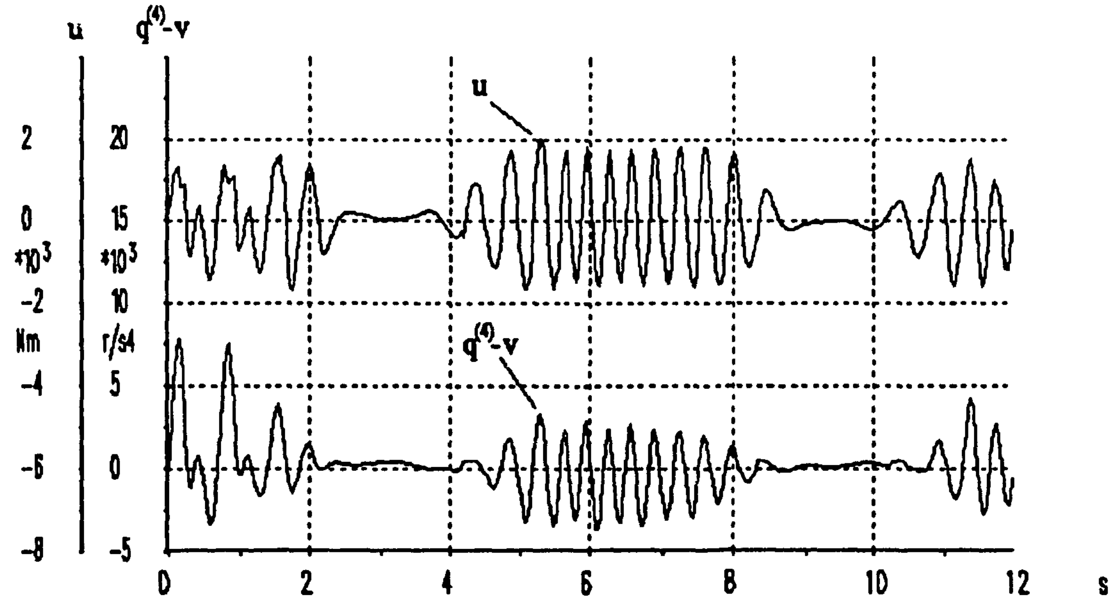


Figure 5.40: Driving torque and modeling error for net trained with more flexible joint.

6
CONCLUSION

By combining several common elements from the fields of robotics and neural networks, we have developed a new control strategy for manipulators with unspecified nonlinear dynamics. The feedback linearization control scheme has been adapted so as to allow the feedback signal to enter the robot nonlinearly, thus eliminating the inherent limitation which makes classical feedback linearization unable to handle certain flexible-joint manipulators [6]. By also replacing the model-based portion of the controller with a neural network, we have obviated the need for any *a priori* knowledge of the robot's dynamical equations, thus meeting our primary design objective.

The emphasis has been on solving the problem of controlling a flexible-joint manipulator, but the approach can be applied directly to other nonlinear control problems. It is best suited for controlling plants whose dynamical equations cannot be derived with sufficient accuracy. In cases where a reliable mathematical model does exist, such a model may well be more desirable than a neural network based strategy for one or more of the following reasons: the mathematical model does not require any training time, it is generally easier to implement, and it may be more accurate than a neural net of tractable size.

The proposed neural network based control strategy has two fundamental limitations. The first is that it presupposes the knowledge of the order of the plant. For plants whose dynamics are unknown, it is highly doubtful that the exact order of the system would be known *a priori*. However, we contend that by observing the plant's open-loop response and examining its physical structure, it is possible to make a fairly accurate estimate of the order. This estimate can be upgraded if neural network training does not show signs of convergence. In either case, the order of the system will be easier to estimate than the complete dynamical equations. The second limitation is that of obtaining a sufficient number of derivatives of the plant output in a physical implementation of the proposed system. This difficulty in obtaining the required state information from the plant is not so much a limitation of our particular strategy, as it is an inherent problem of full-order controller design for high order systems. For the purpose of this work, we have assumed the availability of sufficiently accurate instrumentation to measure the required signals.

The development of our control scheme has required innumerable neural network simulations. As a result of performing these, we have gained much insight into the practical aspects of dealing with neural networks. Although they hold a great deal of promise for solving the large class of problems for which mathematical formulations do not exist, neural nets are not nearly so simple and elegant as they are assumed to be in some of the more theoretical works. Training a net to control a flexible joint has consisted of a long series of "trial-and-error" experiments. With minimal theoretical guidelines, we have had to vary such parameters as network topology, learning rates, initial connection weights, and training signals. Each time a change is made to one such parameter, the entire training procedure is repeated while the net designer watches patiently to see if the most recent change will improve performance as he had hypothesized. The neural network based approach is highly heuristic and often frustrating, but it does open up a path toward attacking those many problems against which conventional mathematical approaches are powerless.

REFERENCES

[1] M. W. Spong and M. Vidyasagar, *Robot Dynamics and Control.* New York: John Wiley and Sons, 1989.

[2] J. J. Craig, *Introduction to Robotics: Mechanics and Control.* Reading, MA: Addison-Wesley, 1986.

[3] A. Isidori, *Nonlinear Control Systems, 2nd Ed.* Berlin: Springer-Verlag, 1989.

[4] A. Isidori, A. J. Krener, C. Gori-Giorgi, and S. Monaco, "Nonlinear decoupling via feedback: A differential geometric approach," *IEEE Trans. Automatic Control*, vol. 26, no. 2, pp. 331–345, 1981.

[5] G. Cesareo and R. Marino, "On the controllability properties of elastic robotics." presented at the 6th International Conference on Analysis and Optimization of Systems, INRIA, (Nice, France), 1984.

[6] R. Marino and S. Nicosia, "On the feedback control of industrial robots with elastic joints: A singular perturbation approach," Tech. Rep. R-84.01, University of Rome, 1984.

[7] M. W. Spong, K. Khorasani, and P. V. Kokotovic, "An integral manifold approach to the feedback control of flexible joint robots," *IEEE Journal of Robotics and Automation*, vol. RA-3, no. 4, pp. 291–300, 1987.

[8] K. J. Astrom and B. Wittenmark, *Adaptive Control.* Reading, MA: Addison-Wesley, 1989.

[9] J. E. Slotine and W. Li, "Adaptive manipulator control: A case study," *IEEE Transaction on Automatic Control*, vol. AC-33, no. 11, pp. 995–1003, 1988.

[10] P. Khosla and T. Kanade, "Parameter identification of robot dynamics," in *Proc. 24th IEEE Conference on Decision and Control*, (Ft. Lauderdale, FL), 1985.

[11] K. Khorasani, "Adaptive control of flexible joint robots," in *1991 IEEE Conference on Robotics and Automation*, (Sacramento, CA), April 1991. To Appear.

[12] B. Widrow, J. R. Glover, J. M. McCool, J. Kaunitz, C. S. Williams, and R. H. Hearn, "Adaptive noise cancelling: Principles and applications," in *Proceedings of the IEEE*, vol. 63, pp. 1692–1716, 1975.

[13] B. Widrow and S. D. Stearns, *Adaptive Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1985.

[14] M. Kawato, Y. Uno, M. Isobe, and R. Suzuki, "Hierarchical neural network model for voluntary movement with application to robotics," *IEEE Control Systems Magazine*, vol. 8, pp. 8–15, April 1988.

[15] J. S. Albus, "A new approach to manipulator control: The cerebellar model articulation control (cmac)," *Trans. ASME, J. Dyn. Syst., Meas. Contr.*, vol. 97, pp. 220–227, Aug. 1975.

[16] W. T. Miller, III, R. P. Hewes, F. H. Glanz, and L. G Kraft, III, "Real-time dynamic control of an industrial manipulator using a neural-network-based learning controller," *IEEE Journal of Robotics and Automation*, vol. 6, pp. 1–9, Feb. 1990.

[17] R. Hecht-Nielsen, "Theory of the backpropagation neural network," in *Proc. Int. Joint Conference on Neural Networks*, vol. I, (Washington D.C.), pp. 593–605, 1989.

[18] J. W. Selinsky and A. Guez, "The role of a priori knowledge of plant dynamics in neurocontroller design," in *Proc. 28th IEEE Conference on Decision and Control*, (Tampa, FL), pp. 1754–1758, 1989.

[19] V. C. Chen and Y. H. Pao, "Learning control with neural networks," in *Proc. IEEE Conference on Robotics and Automation*, (Phoenix, AZ), 1989.

[20] M. Kawato, K. Furukawa, and R. Suzuki, "A hierarchical neural-network model for control and learning of voluntary movement," *Biological Cybernetics*, vol. 57, pp. 169–185, 1987.

[21] D. Psaltis, A. Sideris, and A. A. Yamamura, "A multilayered neural network controller," *IEEE Control Systems Magazine*, pp. 17–21, April 1988.

[22] V. Zeman, R. V. Patel, and K. Khorasani, "A neural network based control strategy for flexible-joint manipulators," in *Proc. 28th IEEE Conference on Decision and Control*, (Tampa, FL), pp. 1759–1764, 1989.

[23] C. F. Stevens, "The neuron," in *The Brain*, pp. 15–27, New York: W. H. Freeman, 1979.

[24] E. R. Kandel, "Small systems of neurons," in *The Brain*, pp. 29–38, New York: McGraw-Hill, 1979.

[25] B. Widrow and M. Hoff, "Adaptive switching circuits," in *1960 IRE WESCON Convention Record, Part 4*, pp. 96–104, Institute of Radio Engineers, 1960.

[26] D. E. Rumelhart and J. L. McClelland and The PDP Research Group, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 1. Cambridge: MIT Press, 1986.

[27] M. L. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. Cambridge: MIT Press, 1969.

[28] R. P. Lippmann, "An introduction to computing with neural nets," *IEEE ASSP Magazine*, vol. 4, pp. 4–22, Apr. 1987.

[29] J. A. Anderson, "Cognitive and psychological computation with neural models," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-13, no. 5, pp. 799–815, 1983.

[30] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," Tech. Rep. 8506, Institute of Cognitive Science, Carnegie-Mellon Univ., 1985.

[31] A. D. Bruce, A. Canning, B. Forest, E. Gardner, and D. J. Wallace, "Learning and memory properties in fully connected networks," in *Neural Networks for Computing* (J. S. Denker, ed.), pp. 65–70, New York: American Institute of Physics, 1986.

[32] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," in *Proc. National Academy of Sciences*, vol. 79, pp. 2554–2558, 1982.

[33] T. Kohonen, *Self Organization and Associative Memory.* New York: Springer-Verlag, 1984.

[34] R. Hecht-Nielsen, "Counter-propagation networks," in *Proc. IEEE First International Conference on Neural Networks*, vol. 2, (San Diego), pp. 19–32, 1987.

[35] C. H. Anderson and E. Abrahams, "A bayesian probability network," in *Neural Networks for Computing* (J. S. Denker, ed.), pp. 7–11, New York: American Institute of Physics, 1986.

[36] R. J. Sasiela, "Forgetting as a way to improve neural-net behaviour," in *Neural Networks for Computing* (J. S. Denker, ed.), pp. 386–391, New York: American Institute of Physics, 1986.

[37] G. E. Hinton, T. J. Sejnowski, and D. H. Ackley, "Boltzmann machines: Constraint satisfaction networks that learn," Tech. Rep. CMU-CS-84-119, Carnegie-Mellon University, 1984.

[38] B. Kosko, "Competitive adaptive bi-directional associative memories," in *Proc. IEEE First International Conference on Neural Networks*, vol. 2, (San Diego), pp. 759–766, 1987.

[39] R. Hecht-Nielsen, "Applications of counterpropagation networks," *Neural Networks*, vol. 1, pp. 131–139, 1988.

[40] K. S. Narendra and K. Parthasarathy, "Identification and control of dynamical systems using neural networks," *IEEE Transactions on Neural Networks*, vol. 1, no. 1, pp. 4–27, 1990.

[41] L. Nguyen, R. V. Patel, and K. Khorasani, "Neural network architectures for the forward kinematics problem in robotics," in *Proc. Int. Joint Conference on Neural Networks*, vol. III, (San Diego), pp. 393–399, 1990.

[42] M. Stinchcombe and H. White, "Approximating and learning unknown mappings using multilayer feedforward networks with bounded weights," in *Proc. Int. Joint Conference on Neural Networks*, vol. III, (San Diego), pp. 7–16, 1990.

[43] J. Wang and B. Malakooti, "On training of artificial neural networks," in *Proc. Int. Joint Conference on Neural Networks*, vol. II, (Washington D.C.), pp. 387–393, 1989.

[44] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 1, ch. 8, pp. 318–362, Cambridge: MIT Press, 1986.

[45] J. M. McInerney, K. G. Haines, S. Biafore, and R. Hecht-Nielsen, "Back propagation error surfaces can have local minima," in *Proc. Int. Joint Conference on Neural Networks*, vol. II, (Washington, D.C.), p. 627, 1989.

[46] L. E. Pfeffer, O. Khatib, and J. Hake, "Joint torque sensory feedback in the control of a PUMA manipulator," *IEEE Journal of Robotics and Automation*, vol. 5, pp. 418–425, Aug. 1989.

[47] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecci, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[48] "NeuralWorks Professional II: Neural Computing." NeuralWare, Inc., Penn Center West, Building IV, Suite 227, Pittsburg, PA 15276.

[49] G. E. Forsythe, M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computations*. Englewood Cliffs: Prentice-Hall, 1977.